PAULO RAFAEL SILVA VIEIRA

DESENVOLVIMENTO E MANUTENÇÃO DE APLICATIVOS MÓVEIS NA VIGNOLI COMUNICAÇÃO LTDA.

LAVRAS – MG

2022

PAULO RAFAEL SILVA VIEIRA

DESENVOLVIMENTO E MANUTENÇÃO DE APLICATIVOS MÓVEIS NA VIGNOLI COMUNICAÇÃO LTDA.

Relatório de estágio supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. DSc. Paulo Afonso Parreira Júnior
Orientador

LAVRAS – MG
2022

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Vieira, Paulo Rafael Silva.
DESENVOLVIMENTO E MANUTENÇÃO DE APLICATIVOS MÓVEIS NA VIGNOLI COMUNICAÇÃO LTDA. / Paulo Rafael Silva Vieira. – Lavras : UFLA, 2022.
41 p. : il.

TCC (graduação)–Universidade Federal de Lavras, 2022.
Orientador: Prof. DSc. Paulo Afonso Parreira Júnior.
Bibliografia.

1. Desenvolvimento de aplicações móveis. 2. Desenvolvimento ágil de software. 3. Desenvolvimento de software. I. Parreira Júnior, Paulo Afonso. II. Título.

CDD-808.066

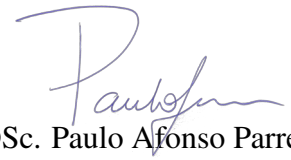
PAULO RAFAEL SILVA VIEIRA

DESENVOLVIMENTO E MANUTENÇÃO DE APLICATIVOS MÓVEIS NA VIGNOLI COMUNICAÇÃO LTDA.

Relatório de estágio supervisionado apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 31 de Agosto de 2022.

Prof. DSc. Júlio César Alves	UFLA
Prof. DSc. Maurício Ronny de Almeida Souza	UFLA



Prof. DSc. Paulo Afonso Parreira Júnior
Orientador

LAVRAS – MG
2022

Dedico à minha família, por todo o apoio que me foi dado durante toda minha vida.

AGRADECIMENTOS

À Universidade Federal de Lavras, por toda a estrutura fornecida.
À Vignoli Comunicação LTDA. e seus funcionários, especialmente os da equipe de desenvolvimento iOS, pela concessão do estágio e por toda a ajuda.
Ao professor DSc. Paulo Afonso Parreira Júnior, pela orientação e por ter sido um excelente professor durante a graduação.
Ao professor DSc. Mauricio Ronny De Almeida Souza, pela orientação no estágio.
A todos os funcionários dos departamentos DCC e DAC da UFLA.
Aos meus amigos, pelo apoio e pelos momentos de lazer e descontração.

RESUMO

O objetivo deste trabalho é apresentar as atividades desenvolvidas na empresa Vignoli Comunicação LTDA, que possui sede em Belo Horizonte e tem como produtos as plataformas *Letras.mus.br*, na área de entretenimento, e *Letras Academy*, na área de educação. Tais plataformas são disponibilizadas como um *website* e também como aplicações móveis para os sistemas *Android* e *iOS*. As atividades do estágio foram realizadas nos dois produtos da Vignoli Comunicação LTDA. para o sistema iOS, e consistiram no desenvolvimento de novas funcionalidades e na manutenção de recursos já implementados. O estágio possibilitou o aprendizado e aplicação de conceitos como *Kanban* e *Scrum*, e o desenvolvimento de *software* utilizando a linguagem de programação *Swift*. Proporcionou, ainda, um aprimoramento das técnicas e conceitos que foram aprendidos durante a graduação bem como uma formação acadêmica mais qualificada.

Palavras-chave: Desenvolvimento de aplicações móveis. Desenvolvimento ágil de *software*.

ABSTRACT

The objective of this work is to present the activities developed in the company Vignoli Comunicação LTDA, which is headquartered in Belo Horizonte and has as products the platforms Letras.mus.br, in the entertainment area, and Letras Academy, in the education area. Such platforms are available as a website and also as mobile applications for Android and iOS systems. The internship activities were carried out in the two products of Vignoli Comunicação LTDA. for the iOS system, and consisted of the development of new functionalities and the maintenance of resources already implemented. The internship allowed the learning and application of concepts such as Kanban and Scrum, and the development of software using the Swift programming language. It also provided an improvement of the techniques and concepts that were learned during graduation as well as a more qualified academic training.

Keywords: Mobile application development. Agile development of software.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Este documento apresenta as atividades desenvolvidas durante o estágio na empresa Vignoli Comunicação Ltda., realizado no período entre 23/02/2021 a 31/07/2021.

A Vignoli Comunicação Ltda. é uma empresa situada na cidade de Belo Horizonte/MG e atua nas áreas de entretenimento e educação. Possui cerca de 80 colaboradores, distribuídos em diferentes áreas de atuação, como moderação de conteúdo, recursos humanos, gestão, *backoffice*, tecnologia da informação, *design*, entre outros.

No campo de entretenimento, tem como produto principal a plataforma *Letras.mus.br*, criada em 2003 e disponibilizada como *website*[1] e como aplicativo para dispositivos com sistemas operacionais Android e iOS. É uma plataforma voltada para o compartilhamento de letras de música e descoberta de novas tendências no mundo musical. Mensalmente, a plataforma recebe cerca de 38 milhões de visitas em seu *website*, ocupando a 48º posição dos 100 sites mais visitados no Brasil (CASAGRANDE, 2022).

Já no campo da educação, a Vignoli Comunicação Ltda. tem como produto a plataforma *Letras Academy*, também disponibilizada como um *website* e como um módulo dentro da aplicação do *Letras.mus.br* para dispositivos móveis. É uma plataforma voltada para o ensino de idiomas por meio da música e, dessa forma, conta com uma seção de cursos de idiomas, uma seção para contratar aulas particulares e uma seção com um dicionário de pronúncia.

Dentre as atividades desempenhadas durante o estágio, destacam-se a manutenção dos recursos existentes no aplicativo *Letras.mus.br* para dispositivos iOS e a implementação da funcionalidade de contratação de professores particulares no módulo do *Letras Academy*, também para o aplicativo iOS.

Este trabalho encontra-se organizado como se segue: o Capítulo 2 apresenta a fundamentação teórica. O Capítulo 3, por sua vez, apresenta as atividades do estágio. Por fim, o Capítulo 4 apresenta as considerações finais deste relatório de estágio.

[1] Disponível em <http:/www.letras.mus.br/>.

2 FUNDAMENTAÇÃO TEÓRICA

A atividade de desenvolvimento consiste em usar uma linguagem de programação, programas específicos e metodologias para produzir, testar e aprimorar *softwares*. Neste capítulo são apresentadas as tecnologias que possibilitaram a execução das atividades no estágio, tais como a linguagem de programação *Swift*, os *frameworks* UIKit e *Foundation*, a IDE (*Integrated Development Environment*) Xcode, o *software* Git e os métodos de desenvolvimento de software *Kanban* e *Scrum*.

2.1 A linguagem de programação *Swift*

Swift é uma linguagem de programação desenvolvida pela *Apple* e disponibilizada como um projeto *open-source*. Possui interoperabilidade com a linguagem *Objective-C* e compatibilidade com a linguagem C, sendo então pertencente à família C. É suportada por diversos sistemas operacionais, sendo eles iOS, macOS, watchOS, tvOS, Linux e *Windows*. É uma linguagem compilada, com tipagem forte, caracterizada como estática, porém com inferência de tipagem. Pode ser considerada como uma linguagem multi-paradigma, suportando os paradigmas orientado a objetos, imperativo, concorrente e funcional (APPLE, 2022b). A Figura 2.1 mostra um exemplo de um programa que exibe o texto “Hello, Paulo” no *console*.

Figura 2.1 – Exemplo de um programa simples em *Swift*

```
func greetPerson(name: String) -> Void {
    print("Hello, \(name)")
}

greetPerson(name: "Paulo")
```

Fonte: Autor

Já a Figura 2.2 mostra um exemplo de um *struct* “Carro”, um *enum* “Cor”, uma classe “Pessoa” e uma classe “Aluno” que herda da classe “Pessoa”, implementados na linguagem *Swift*. Neste exemplo, é possível ver diferentes tipos de dados e permissões de acesso que a linguagem oferece para o desenvolvedor.

Na primeira linha, o *framework Foundation* é importado, o qual é responsável por fornecer algumas funcionalidades básicas como manipulação de *strings*, datas, ordenação, chamadas de rede, dentre outras. Dentro das estruturas de dados, é possível ver duas formas de criação de dados que a linguagem fornece: *var* é usado para criar um valor variável, enquanto *let* é usado

para criar um valor constante. Também é possível ver alguns dos diferentes tipos de acesso: *private*, utilizado para definir um campo no qual tanto o *getter* quanto o *setter* são privados, *private(set)*, usado para definir um campo no qual apenas o seu *setter* é privado, *internal*, usado para definir que um campo é acessível apenas para o módulo onde ele se encontra e *public*, usado para definir que um campo é acessível para todos os módulos.

Figura 2.2 – Exemplo de tipos de dados na linguagem *Swift*

```
import Foundation

struct Carro {
    enum Cor {
        case vermelho
        case azul
        case prata
        case preto
    }

    private let marca: String = "Ford"
    private let modelo: String = "Ka"
    private(set) var placa: String = "ABC-1234"
    var cor: Cor = .vermelho
}

class Pessoa {
    internal let nome: String = "Caio da Silva"
    private let cpf: String = "111.111.111-11"
    private(set) var peso: Float = 60.7
    public var altura: Float = 1.78
}

class Aluno: Pessoa {
    private let numeroDeMatricula: Int = 123456
    private(set) var notas: [Float] = [10.8, 12.0, 3.5, 7.8]
}
```

Fonte: Autor

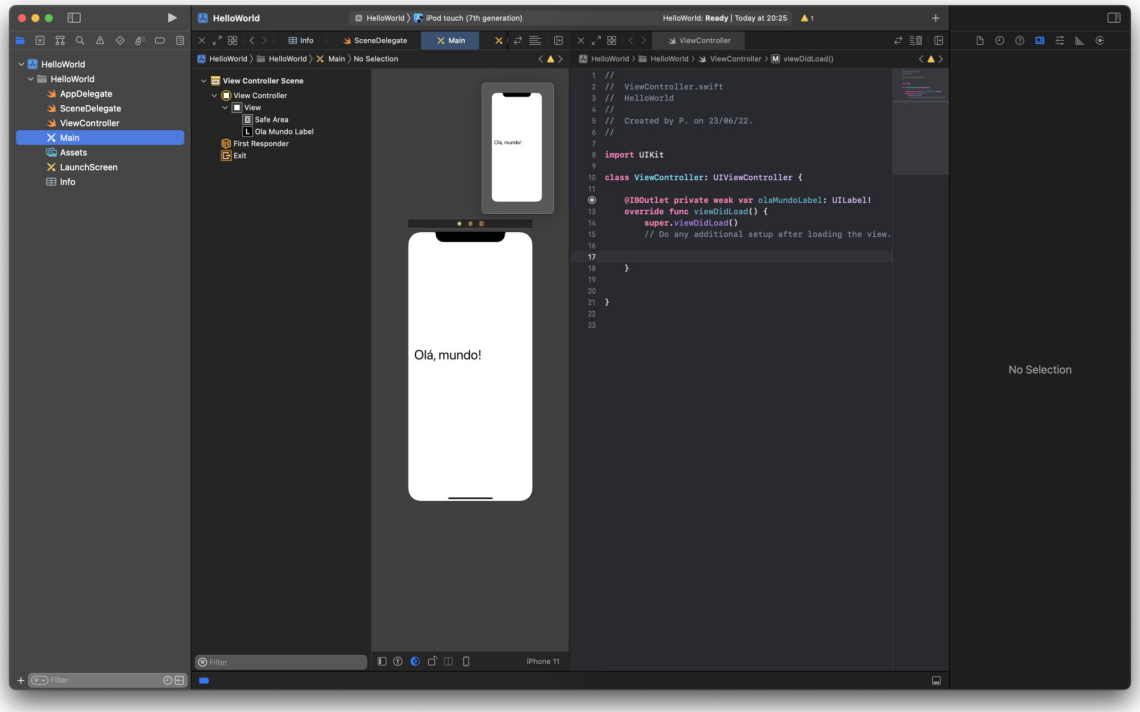
No estágio, a linguagem *Swift* foi utilizada para o desenvolvimento de todas as tarefas, tendo em vista que o aplicativo *Letras.mus.br*, bem como seu módulo *Letras Academy*, possuem *Swift* como linguagem de programação principal.

2.2 A IDE Xcode

Xcode é um ambiente de desenvolvimento integrado criado pela *Apple* para computadores com sistema operacional macOS. É a única ferramenta com suporte oficial para criação

de aplicações nativas destinadas aos sistemas iOS, tvOS, watchOS e macOS. No processo de desenvolvimento de uma aplicação, o Xcode é utilizado tanto para a criação de *layouts*, quanto para a escrita de código (APPLE, 2022c). A Figura 2.3 apresenta a *interface* padrão do Xcode com um projeto autoral do sistema iOS aberto.

Figura 2.3 – Interface do Xcode



Fonte: Autor

No canto esquerdo, é possível ver a seção de navegação, onde são listados todos os arquivos e dependências que fazem parte do projeto criado. Na parte central, encontra-se o editor, que na imagem está dividido em dois. Na esquerda da seção central, há um arquivo conhecido como *storyboard*, no qual é possível criar o *layout* das telas do *software*. Na direita da seção central, há um editor de texto aberto, por meio do qual é possível codificar o comportamento do aplicativo. Já no canto mais à direita encontra-se a seção de utilidades, que permite ao desenvolvedor alterar propriedades do editor ou arquivo que está em foco, consultar documentações e fazer ligações entre componentes visuais e propriedades dos arquivos de código-fonte.

No estágio realizado, todas as atividades de codificação e desenvolvimento de *layout* foram efetuadas utilizando o Xcode.

2.3 O *framework Foundation*

Foundation é um *framework* criado pela *Apple* que complementa a biblioteca padrão da linguagem *Swift*. Possui funções, estruturas de dados e protocolos que facilitam a manipulação de *strings*, manipulação de datas, *networking*, filtro de dados, acesso ao sistema de arquivos, dentre outras, além de fornecer a classe base que faz parte da hierarquia da maioria das classes em *Swift* e *Objective-C* (APPLE, 2022a). A Figura 2.4 mostra um exemplo de código em *Swift* que utiliza o *framework Foundation*.

No exemplo, o *framework* é incluído para se ter acesso ao protocolo *Decodable*, que sinaliza que um modelo pode ser decodificável de alguma forma; para se ter acesso ao objeto *URLSession*, responsável por fazer uma requisição à uma API; e para se ter acesso ao objeto *JSONDecoder*, responsável por decodificar a resposta da API de acordo com o modelo decodificável.

Figura 2.4 – Exemplo de programa utilizando o *Foundation*

```
import Foundation

struct ChuckNorrisJoke: Decodable {
    let id: String
    let url: String
    let value: String
}

func fetchJoke() async throws -> ChuckNorrisJoke {
    let apiURL: URL! = URL(string:
       "https://api.chucknorris.io/jokes/random")!
    let (data, _) = try await URLSession.shared.data(from: apiURL)
    let decodedResult: ChuckNorrisJoke = try
       JSONDecoder().decode(ChuckNorrisJoke.self, from: data)

    return decodedResult
}

Task {
    do {
        let joke: ChuckNorrisJoke = try await fetchJoke()
        print(joke.value)
    } catch {
        print("Uma exce  o foi lan ada:
           \(error.localizedDescription)")
    }
}
```

Fonte: Autor

2.4 O *framework UIKit*

UIKit é um *framework* criado pela *Apple* para a construção de *interfaces* do usuário e a definição de comportamentos dos componentes da aplicação para o sistema operacional iOS. A criação de *layouts* com o *framework UIKit* pode ser feita principalmente de forma visual, por meio de arquivos conhecidos como *Storyboards*.

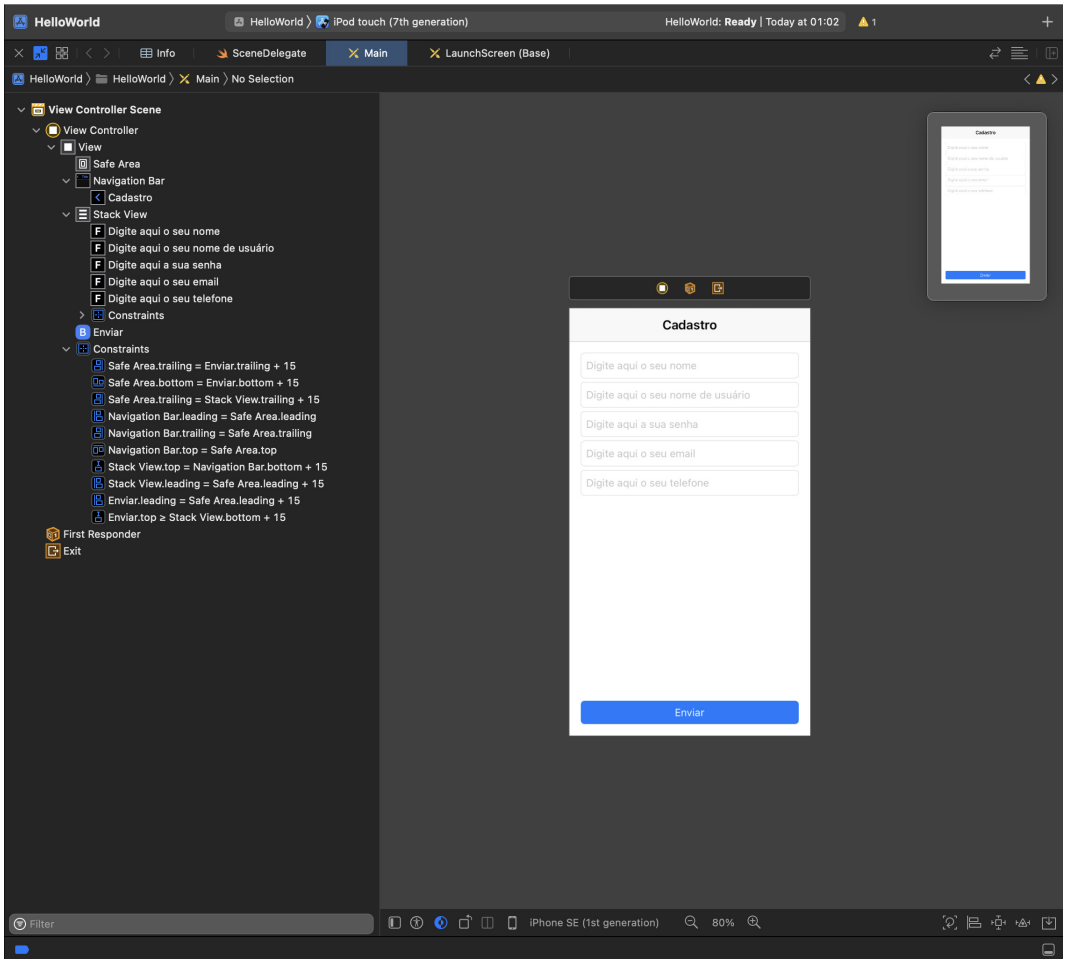
Nos arquivos de *Storyboard*, é possível adicionar em uma tela componentes como caixas de texto, imagens, dentre outros, de forma a criar outros componentes. Tais componentes são, na verdade, objetos que herdam de uma classe chamada *UIView*, um objeto que é responsável por exibir algum conteúdo na tela da aplicação. Cada tela é formada por um conjunto de *UIView* e possui um objeto chamado *UIViewController*, que controla todas as ações, eventos, conexões entre componentes e código, o estado dos dados e a hierarquia de componentes.

Na Figura 2.5 é possível ver o *Layout* de uma simples tela de cadastro, criada por meio de um arquivo *Storyboard*. Para formar a tela de cadastro, foram usados componentes de entrada de texto, que representam um objeto chamado *UITextField*, um componente de um botão, que representa um objeto chamado *UIButton*, e uma barra de navegação, que representa um objeto *UINavigationBar*.

Tais componentes podem ser conectados com variáveis no código *Swift*. Para fazer essa conexão entre componente e código, o *framework UIKit* disponibiliza duas propriedades de variáveis e métodos: *IBOutlet* e *IBAction*. *IBOutlets* fazem a conexão entre um componente e uma variável no código, permitindo que propriedades visuais possam ser alteradas por meio de codificação. Já as *IBActions* fazem a conexão entre uma ação na interface do usuário, como o toque em um botão, e uma função no *UIViewController* responsável por aquela tela.

A Figura 2.6 demonstra como seria um arquivo *UIViewController* que controla a tela de cadastro da Figura 2.5. Nas linhas iniciais da classe *CadastroViewController*, são definidos alguns *IBOutlets* referentes aos componentes que foram adicionados no arquivo *Storyboard*. No método *viewDidLoad*, o texto de cada campo de texto é alterado. Então, assim que a tela terminar de carregar, o valor de cada campo de texto já estará preenchido. Por fim, o método *enviarButtonWasTapped* é definido como um *IBAction*, pois está conectado com o botão de enviar do *layout*. Quando ele for pressionado, o texto “Botao foi pressionado” será mostrado no console.

Figura 2.5 – *Layout* criado por meio de um arquivo *Storyboard*



Fonte: Autor

Figura 2.6 – *UIViewController* referente ao *layout* da tela de cadastro

```
import UIKit

class CadastroViewController: UIViewController {
    @IBOutlet private weak var cadastroNavigationBar:
       UINavigationBar!
    @IBOutlet private weak var nomeTextField: UITextField!
    @IBOutlet private weak var nomeDeUsuarioTextField: UITextField!
    @IBOutlet private weak var senhaTextField: UITextField!
    @IBOutlet private weak var emailTextField: UITextField!
    @IBOutlet private weak var telefoneTextField: UITextField!
    @IBOutlet private weak var enviarButton: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()

        nomeTextField.text = "Paulo"
        nomeDeUsuarioTextField.text = "paulorsv01"
        senhaTextField.text = "hunter2"
        emailTextField.text = "paulorsv01@abc.com"
        telefoneTextField.text = "+55 (11) 99999-9999"
    }

    @IBAction private func enviarButtonWasTapped(_ sender: UIButton)
       {
        print("Botao foi pressionado.")
    }
}
```

Fonte: Autor

2.5 O *software* Git

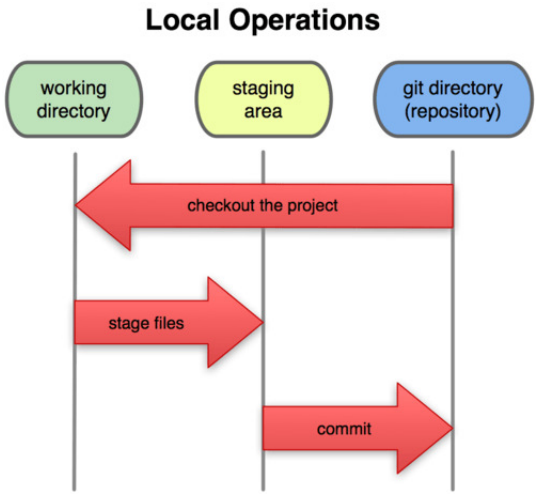
Git é um sistema de controle de versionamento distribuído desenvolvido por Linus Torvalds em 2005. É utilizado para acompanhar e coordenar mudanças feitas em arquivos de um projeto, gerando um histórico de alterações e permitindo que diferentes versões de um projeto possam ser acessadas de maneira simples (CHACON; STRAUB, 2014).

Ele tem como base dois procedimentos: *add* e *commit*. Quando algum arquivo sofre uma modificação, ele fica marcado como um arquivo *unstaged*, ou seja, um arquivo cujas alterações ainda não fazem parte das modificações que serão salvas pelo procedimento de *commit*. O procedimento *add* faz uma marcação, deixando o arquivo como *staged*. Uma vez que as modificações estão marcadas como *staged*, o procedimento *commit* salva essas modificações no repositório *git* e cria um *snapshot* (como se fosse uma fotografia do arquivo em determinado momento no tempo) referenciado por um código *hash* único. Com este *hash*, é possível consultar como era esse *snapshot*, mesmo que mudanças mais recentes já tenham sido adicionadas no repositório. A Figura 2.7 mostra como funciona este fluxo de trabalho usando o Git.

Outras funcionalidades do *software* também foram utilizadas durante a execução das atividades do estágio, como o envio de *commits* para um repositório remoto por meio do procedimento *push*, a criação de *branches*, a junção de *commits* feito por outros desenvolvedores por meio dos procedimentos *merge* e *rebase* e o *download* de modificações presentes em um repositório remoto por meio do procedimento *pull*. Todas as atividades efetuadas no estágio foram feitas usando o Git para versionamento e os repositórios utilizados foram hospedados por meio da plataforma *GitHub*[1].

[1] Disponível em <https:/www.github.com/>

Figura 2.7 – Operações de um fluxo de trabalho usando o Git



Fonte: (CHACON; STRAUB, 2014)

2.6 *GraphQL*

GraphQL é uma linguagem de consulta de dados *open-source*, criada em 2012 pelo *Facebook*. É uma linguagem de consultas hierárquica, ou seja, os objetos conseguem se relacionar entre si permitindo que de um objeto seja possível alcançar outro, fortemente tipada e criada para ser uma alternativa ao modelo *REST* (BYRON, 2015).

Diferentemente de outras formas de efetuar requisições, o *GraphQL* permite que de um determinado ponto de comunicação sejam selecionados apenas os campos desejados no retorno de uma consulta, e permite que campos sejam consultados de forma aninhada. Tal operação tem o de *query*, e é similar a operação de *GET* do modelo *REST*. Uma diferença está no fato de que na operação *GET* os campos que estarão presentes no retorno são fixos.

Para a modificação de dados é utilizado o conceito de *mutations*, operações que recebem parâmetros e que efetuam modificações no lado do servidor. Normalmente, uma mutação em *GraphQL* recebe alguns parâmetros de entrada, faz um procedimento e no final pode ou não retornar algum campo, que também pode ter outros campos dentro de sua estrutura. Uma das vantagens dessa abordagem é que os dados retornados funcionam exatamente como dados obtidos por meio da operação de consulta, sendo possível utilizar estes dados para atualizar objetos na aplicação. É uma operação que pode funcionar como as operações *PUT*, *DELETE* e *POST* do modelo *REST*.

As Figuras 2.8 e 2.9 apresentam uma consulta e uma mutação feitas em *GraphQL*, respectivamente.

Figura 2.8 – *Query* em GraphQL para consultar todos os animais de um banco de dados

```
query GetAllPets {
  pets {
    name
    petType
  }
}
```

Fonte: (STEMMLER, 2021)

Figura 2.9 – *Mutation* em GraphQL para adicionar um novo animal no banco de dados

```
mutation AddNewPet ($name: String!, $petType: PetType) {
  addPet(name: $name, petType: $petType) {
    id
    name
    petType
  }
}
```

Fonte: (STEMMLER, 2021)

Na Vignoli Comunicação LTDA., a API principal do *Letras Academy* é implementada utilizando a linguagem de consulta *GraphQL*.

2.7 Scrum

O *Scrum* é um sistema de gerenciamento de projetos e um modelo de desenvolvimento ágil com um formato iterativo e incremental. Cada ciclo no Scrum é conhecido como *Sprint*, um processo com uma duração de tempo fixa e com um conjunto de tarefas que devem ser implementadas nesse período. No final de cada *Sprint*, uma nova versão com novas funcionalidades e melhorias do sistema é lançada (SCHWABER; SUTHERLAND, 2020).

No Scrum, antes do início de uma *Sprint* ocorre uma reunião de planejamento chamada *planning*. Nela, o *product owner* adiciona no *backlog* da *sprint* as tarefas do *backlog* do produto que serão desenvolvidas naquela *Sprint*. No modelo Scrum, o *backlog* consiste em uma lista com todas as funcionalidades e recursos planejados para o produto, sendo que normalmente ele é controlado pelo *product owner*, pessoa responsável por definir prioridade das tarefas que estão neste *backlog* e planejar o lançamento de novas funcionalidades, melhorias e correções. Outra figura-chave no processo do Scrum é o *Scrum Master*, pessoa responsável por garantir que os conceitos da metodologia estão sendo aplicados de maneira correta e que a equipe detém do conhecimento destes conceitos.

Na reunião de planejamento, também é estimada a “dificuldade” de cada tarefa, permitindo que o andamento da *Sprint* seja calculado de forma simples. Além da *planning*, outra reunião que ocorre é a *daily*, uma reunião diária, na qual problemas encontrados e informações importantes são discutidos. No final da *Sprint*, ocorre a reunião de retrospectiva. Nela, as tarefas completadas são arquivadas, as tarefas incompletas são rotuladas para serem adicionadas na próxima *Sprint* e as dificuldades que foram encontradas ao longo da *Sprint* são debatidas. Apesar de não acontecer na Vignoli Comunicação LTDA., o modelo Scrum prevê uma reunião após o término de cada período de *sprint* denominada *Sprint Review Meeting*, onde são apresentadas para as partes interessadas as atividades que foram efetuadas e o resultado alcançado durante o processo.

Na Vignoli Comunicação LTDA., tanto o *product backlog* quanto o *sprint backlog* seguem o formato de uma planilha *online*. Nela, é possível encontrar informações como a prioridade da tarefa, quem é o responsável por ela, qual é o seu *status*, quais são os *testers* da tarefa, qual é a descrição do que tem que ser feito, entre outras informações. A Figura 2.10 ilustra como era a planilha usada durante o estágio, para o desenvolvimento das tarefas.

Figura 2.10 – Planilha com dados de uma *Sprint*

		Status	Owner	Tester 1	Tester 2	Description	Github Issues	A referencia
						Academy - Home		
0		review	Gilmar		Caio	Criar o conteudo das outras 2 abas	https://github.com/StudioSol/iOSLetras/issues/4062	18
0		doing	Paulo	Gilmar		Infoview para solicitar update do app	https://github.com/StudioSol/iOSLetras/issues/4055	18
						Academy - Home do aluno		
0		aguardando				Exibir views de pagamento pendente Aguardando issue https://github.com/StudioSol/LetrasAcademy/issues/775	https://github.com/StudioSol/iOSLetras/issues/4002	0
0		todo				Home do aluno faltando informações ao abrir o app	https://github.com/StudioSol/iOSLetras/issues/4074	
0		aguardando				Correção do isTrialClassMadeBySubscription	https://github.com/StudioSol/iOSLetras/issues/4153	
						Listagem de professores		
0		todo				Academy crashando na listagem de professores, rodando pelo Letras e desconectando pelas configurações	https://github.com/StudioSol/iOSLetras/issues/4159	
						Perfil do professor		
0		aguardando				Exibir views de pagamento pendente Aguardando issue https://github.com/StudioSol/LetrasAcademy/issues/775	https://github.com/StudioSol/iOSLetras/issues/4004	0
-2		todo				Navigation bar no iOS 11 com altura pequena	https://github.com/StudioSol/iOSLetras/issues/4119	
						Agendamento		
0		aguardando	Caio			2 loadings e um não some após o agendamento de aulas Aguardando correção da API para testar	https://github.com/StudioSol/iOSLetras/issues/4118	

Fonte: Vignoli Comunicação LTDA, 2022.

No estágio, o Scrum foi usado como modelo de desenvolvimento por alguns meses, sendo substituído pelo modelo Kanban após um tempo.

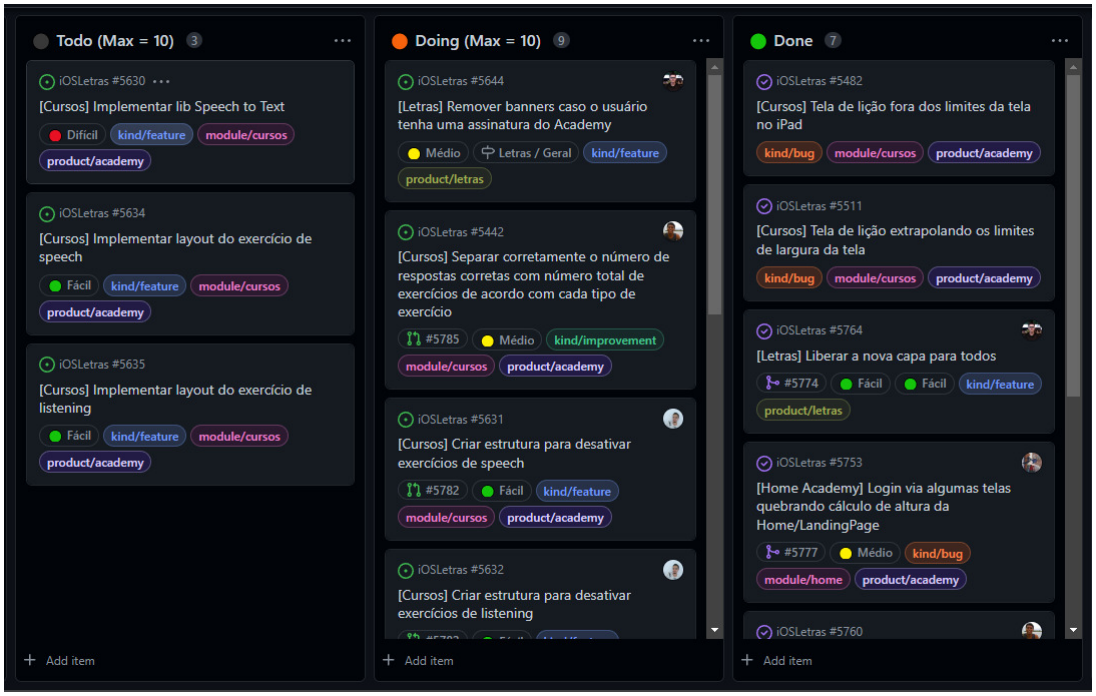
2.8 *Kanban*

Kanban é um sistema de desenvolvimento japonês criado pela Toyota na década de 60, o qual é implementado com base em três componentes: o quadro, as colunas e os cartões. Os cartões representam o que precisa ser feito, as colunas representam o estado do cartão e o quadro é o que faz a organização dos cartões nas colunas (MENDES, 2020).

No modelo *Kanban* implementado na Vignoli Comunicação LTDA., as reuniões de planejamento que antes aconteciam no modelo *Scrum* deixaram de acontecer, pois a equipe julgou que tais reuniões demandavam muito tempo e muitas informações discutidas se perdiam alguns dias após a reunião. A falta de uma pessoa no papel de *Scrum Master* e o tamanho da equipe também contribuíram para a troca de metodologias.

No lugar da planilha, passou a ser usado um quadro de *Kanban* na plataforma *GitHub*, com as colunas: *To-Do*, *Doing* e *Done*. Novas tarefas eram adicionadas na seção de *To-do* do quadro *Kanban* nas reuniões diárias e, tanto as informações importantes quanto as dificuldades encontradas nas tarefas em *doing*, eram discutidas nessa mesma reunião. No final de cada mês, ocorria a reunião de *review*, na qual várias estatísticas de como foi o andamento das tarefas eram apresentadas. A Figura 2.11 apresenta o quadro *Kanban* utilizado no desenvolvimento das tarefas. As tarefas efetuadas durante o estágio foram distribuídas e acompanhadas pelo modelo de desenvolvimento *Kanban*.

Figura 2.11 – Quadro *Kanban* utilizado no desenvolvimento das tarefas



Fonte: Vignoli Comunicação LTDA, 2022.

3 ATIVIDADES REALIZADAS

A empresa Vignoli Comunicação LTDA. atua no setor de entretenimento com a plataforma Letras.mus.br, já consolidada, e durante o período do estágio a empresa estava lançando a plataforma *Letras Academy*. Em relação ao *Letras.mus.br*, as atividades consistiram na manutenção de recursos existentes. Já em relação ao *Letras Academy*, as atividades desenvolvidas contemplaram o desenvolvimento de novas funcionalidades.

Neste capítulo são descritas algumas das atividades que foram realizadas durante o estágio, tanto na aplicação do *Letras.mus.br* quanto no módulo de contratação de professores do *Letras Academy*. A seleção das atividades se deu por meio da sua relevância em relação aos produtos da empresa.

3.1 Criação de estrutura para desativar serviços de *streaming* na aplicação *Letras.mus.br*

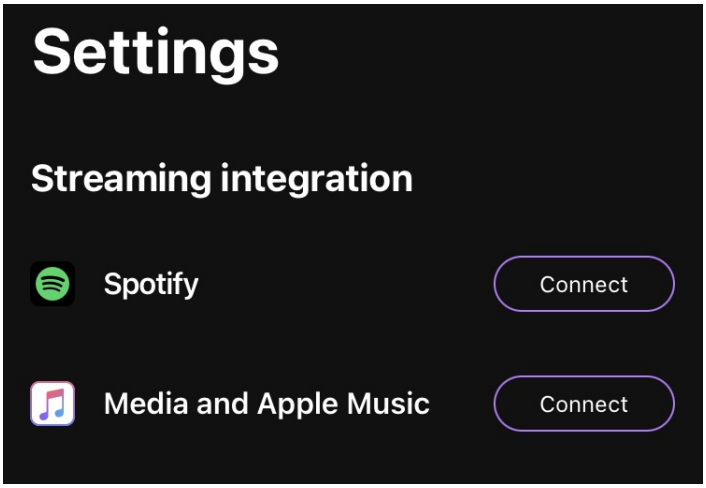
Com o lançamento de novas atualizações do sistema operacional iOS, novas funcionalidades e melhorias em funcionalidades já existentes são apresentadas, porém, muitas vezes, esses recursos não são fornecidos de maneira retroativa. Assim, torna-se necessário aumentar a versão mínima requerida para o funcionamento da aplicação. Um problema que se manifesta ao efetuar este procedimento é a impossibilidade de lançar atualizações com correções de *bugs* e novas melhorias para os dispositivos com versão de sistema abaixo da versão mínima requerida.

O aplicativo *Letras.mus.br* possui como funcionalidade a integração com alguns serviços de *streaming*, como *Spotify*, *Apple Music* e *YouTube*. Tal funcionalidade é implementada utilizando o SDK (*Software Development Kit*, ou kit de desenvolvimento de software) fornecido pela plataforma de *streaming* como uma biblioteca já compilada. A Figura 3.1 expõe as plataformas de *streaming* que são implementadas dentro da aplicação do *Letras.mus.br*.

Como era necessário aumentar a versão mínima requerida da aplicação para ter acesso às novas funcionalidades e isso impedia o lançamento de atualizações para quem não pudesse atualizar o sistema operacional, foi preciso criar uma estrutura que permitisse desabilitar, de forma remota, a integração com alguma plataforma caso algum problema surja na biblioteca fornecida. Embora haja dados[1] que indiquem uma rápida adoção de novas versões do sistema pelos usuários após o lançamento, a Vignoli Comunicação Ltda. optou por implementar uma funcionalidade para mitigar problemas que poderiam surgir. Tal funcionalidade foi implemen-

[1] <https://developer.apple.com/support/app-store/>

Figura 3.1 – Plataformas de *streaming* de áudio integradas no *Letras.mus.br*

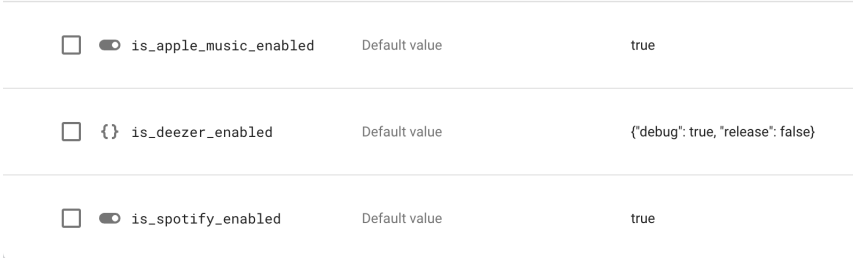


Fonte: Vignoli Comunicação LTDA, 2022

tada por meio do sistema *Remote Config* do *framework Firebase*, que possibilita a criação de um dicionário com pares de chave-valor e que é atualizado de maneira constante durante a execução da aplicação.

Com o dicionário e seus valores, é possível alterar comportamentos e configurações sem a necessidade de que uma atualização seja lançada, sendo necessário apenas alterar o valor que a chave guarda e configurar a aplicação para usar o valor buscado pelo *framework*. A Figura 3.2 apresenta uma estrutura criada de configuração remota na plataforma *Firebase* similar à estrutura criada para a solução desta atividade, e a Figura 3.3 mostra um trecho de código em *Swift* que faz a leitura dos dados da figura anterior.

Figura 3.2 – Estrutura de configuração remota na plataforma *Firebase*



Fonte: Autor

Figura 3.3 – Código em *Swift* para leitura de um valor do *Firebase*

```
import UIKit
import FirebaseRemoteConfig

class ViewController: UIViewController {
    static private let spotifyKeyValue: String = "is_spotify_enabled"
    @IBOutlet private weak var spotifyValue: UILabel!
    private var remoteConfig: RemoteConfig!
    private var isSpotifyActive: Bool = false

    override func viewDidLoad() {
        super.viewDidLoad()
        remoteConfig = RemoteConfig.remoteConfig()
        let settings = RemoteConfigSettings()
        settings.minimumFetchInterval = 0
        remoteConfig.configSettings = settings

        spotifyValue.text = "Spotify est  ativo? \(isSpotifyActive)"
        fetchAndSetupValues(completionHandler: {
            self.spotifyValue.text = "Spotify est  ativo?
               \(isSpotifyActive)"
        })
    }

        func fetchAndSetupValues(completionHandler: @escaping () ->
           Void) -> Void {
            remoteConfig.fetchAndActivate { (status, error) -> Void
               in
                if status == .success {
                    isSpotifyActive = remoteConfig.configValue(for:
                       Self.spotifyKeyValue).boolValue as! Bool
                    completionHandler()
                }
            }
        }
    }
}
```

Fonte: Autor

No exemplo de código da Figura 3.3, a biblioteca do *RemoteConfig* é inicializada e configurada dentro do método *viewDidLoad* e a *label spotifyValue* tem seu texto configurado para exibir o texto “Spotify está ativo? *false*”. Após isso, é feita uma busca e ativação dos valores remotos no método *fetchAndSetupValues*. Tal método recebe uma função anônima como parâmetro que será executada após os valores remotos estarem salvos. Quando a função anônima for executada, a variável *isSpotifyActive* guardará um valor verdadeiro e então o texto da *label spotifyValue* será alterado para “Spotify está ativo? *true*”.

Das dificuldades encontradas, destacam-se a ausência de conhecimento da estrutura interna do código-fonte da aplicação *Letras.mus.br* e do uso do sistema de configuração remota do *framework Firebase*.

3.2 Atividades desenvolvidas no módulo do *Letras Academy*

Nesta seção, são descritas algumas atividades que foram realizadas dentro do módulo do *Letras Academy*, bem como as dificuldades encontradas e como elas foram solucionadas.

3.2.1 Criação de estrutura para aplicação de múltiplos efeitos de sombra em componentes

Para a criação dos *layouts* das telas e dos componentes do módulo de professores do *Letras Academy*, uma abordagem utilizada pela equipe de *design* foi a criação de um *design system*[2], o qual englobava todos os componentes utilizados, todas as cores da aplicação, toda a tipografia utilizada e todas as sombras que eram aplicadas nos componentes.

A tarefa realizada pelo estagiário consistiu na criação de uma estrutura que implementava os efeitos de sombra padronizados no *design system*. A Figura 3.4 demonstra como utilizar e quais são as propriedades que podem ser alteradas no *framework UIKit* para aplicar uma sombra simples em um componente, sendo elas a cor, o deslocamento, o raio e a opacidade. Já a Figura 3.5 apresenta o resultado do *layout* do código apresentado na figura anterior.

Figura 3.4 – Exemplo de aplicação de sombra utilizando UIKit

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet private weak var shadowView: UIView!
    override func viewDidLoad() {
        super.viewDidLoad()
        shadowView.layer.shadowColor = UIColor.black.cgColor
        shadowView.layer.shadowOffset = CGSize(width: 0, height: 4)
        shadowView.layer.shadowRadius = 4
        shadowView.layer.shadowOpacity = 0.4
    }
}
```

Fonte: Autor

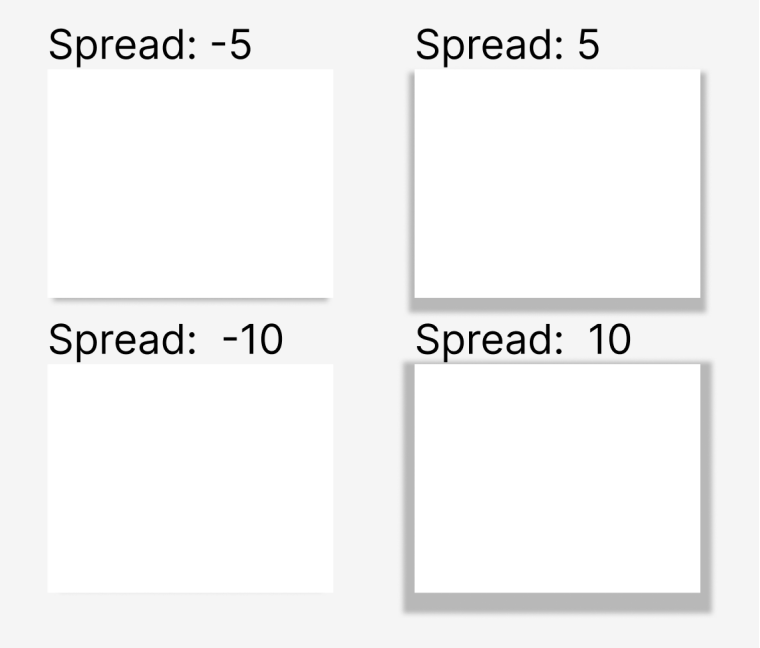
[2] *Design System* é um documento que agrupa um conjunto de elementos reutilizáveis e que seguem um mesmo padrão, usado como um guia para construção de *layouts* e identidade visual (FESSENDEN, 2021)

Figura 3.5 – Resultado de código de exemplo de aplicação de sombra em componente

Fonte: Autor

Em alguns componentes do *design system*, eram previstos múltiplos efeitos de sombra com a finalidade de produzir um efeito de elevação do componente, permitindo que o usuário consiga notar a hierarquia dos diferentes elementos presentes na tela. Além disso, também era previsto que os efeitos de sombra tivessem uma propriedade adicional que controla o seu *spread*, ou espalhamento, propriedade esta que controla quão maior ou menor a sombra é com relação ao componente que a possui. A Figura 3.6 apresenta como diferentes valores para o espalhamento afetam a sombra de um componente.

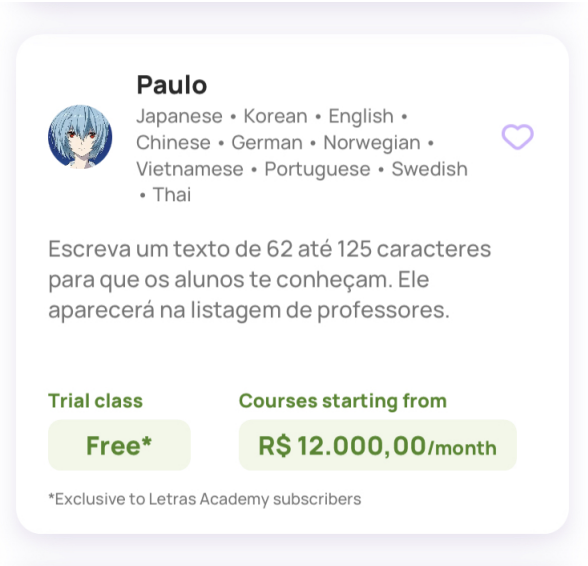
Figura 3.6 – Efeito de valores distintos para o *spread* de uma sombra



Fonte: Autor

Uma vez que o *framework UIKit* não disponibiliza uma forma nativa de aplicar mais de um efeito de sombra por componente, a criação da estrutura se mostrou trabalhosa, sendo preciso pensar em formas alternativas de conseguir o efeito desejado. Outra dificuldade encontrada foi na aplicação da propriedade de *spread* de sombra, que também não possui uma forma de ser aplicada por meio da interface disponibilizada. Assim, foi necessário utilizar componentes vazios debaixo do componente original para desenhar mais de uma sombra e definir o tamanho e formato das sombras de forma manual para que o efeito ficasse de acordo com o *layout* previsto pelos *designers*. A Figura 3.7 apresenta o resultado final dos efeitos aplicados por meio da estrutura criada.

Figura 3.7 – Sombra aplicada no *Letras Academy* por meio da estrutura criada



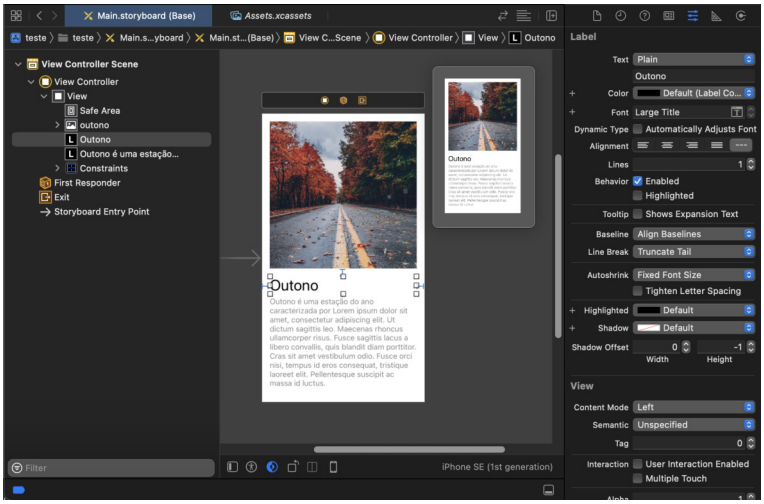
Fonte: Vignoli Comunicação LTDA, 2022

3.2.2 Ajustes pontuais no *layout* de telas e componentes

Como a aplicação estava em constante evolução, várias tarefas efetuadas pelo estagiário eram tarefas menores, com o propósito de ajustar tanto o visual quanto os textos de alguns componentes que já estavam implementados.

Essas modificações, muitas vezes, eram feitas nos arquivos de *Storyboard*, pois são arquivos que agrupam várias configurações de *views* e permitem a alteração dessas configurações de maneira rápida e fácil, sendo necessário apenas selecionar o componente e usar a seção de inspeção de atributos, apresentada na Figura 3.8.

Figura 3.8 – Arquivo do tipo *Storyboard* com seção de inspeção de atributos selecionada



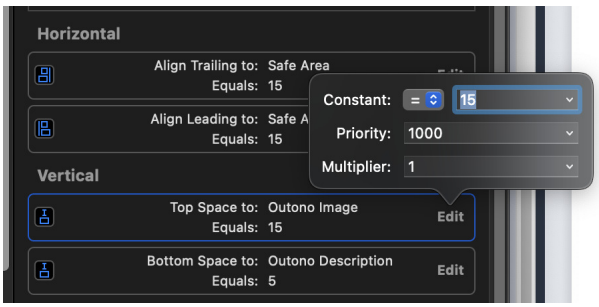
Fonte: Autor

Outro ponto que os arquivos de *Storyboard* permitem ajustar com facilidade é o espaçamento entre componentes, por meio de um sistema chamado *auto-layout*, disponível no *framework UIKit*. Tal sistema utiliza de *constraints*, palavra em inglês que significa restrições, para criar regras de como um componente deve ser portar com relação ao seu tamanho e com relação a outros componentes. Isso permite criar regras específicas para cada tamanho de tela, de forma que o *layout* fique coerente tanto em dispositivos pequenos quanto em dispositivos grandes. A Figura 3.9 apresenta algumas das regras que podem ser criadas, como por exemplo o espaçamento horizontal, o espaçamento vertical, a distância entre um componente e outro no topo, se o componente será centralizado, dentre outras. Já a Figura 3.10 apresenta as propriedades que podem ser alteradas para uma determinada regra.

Figura 3.9 – Algumas regras de *constraint* que podem ser adicionadas pelo *UIKit*

Fonte: Autor

Figura 3.10 – Propriedades modificáveis de regras de *constraint*



Fonte: Autor

Usando destas funcionalidades dos arquivos do tipo *Storyboard*, os ajustes principais que foram efetuados foram substituição de componentes, alteração de propriedades como cores, sombras, textos, e também alteração em regras de espaçamento. As dificuldades encontradas

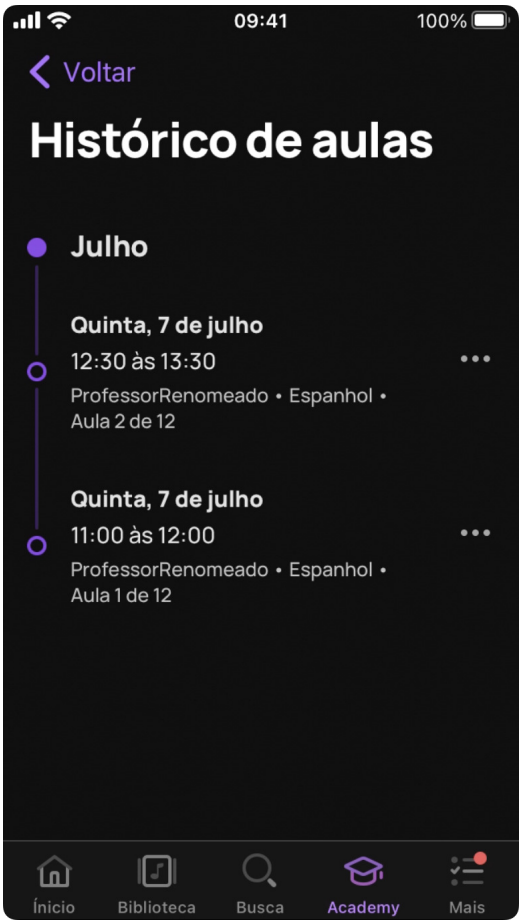
eram provenientes das regras de *constraints*, pois, muitas vezes, era necessário atualizá-las ou removê-las sem modificar indevidamente as outras regras que seriam mantidas.

3.2.3 Implementação de telas de erro

Outra tarefa realizada pelo estagiário se deu na implementação de telas de erro para a seção de histórico de aulas de um aluno, quando, por exemplo, não há conexão ou quando há um erro no servidor.

A implementação da parte visual das telas de erro se deu por meio do *framework UIKit*, utilizando de um arquivo do tipo *Storyboard* que continha as telas do histórico de aulas, apresentada na Figura 3.11.

Figura 3.11 – Tela de histórico de aulas do Letras *Academy*

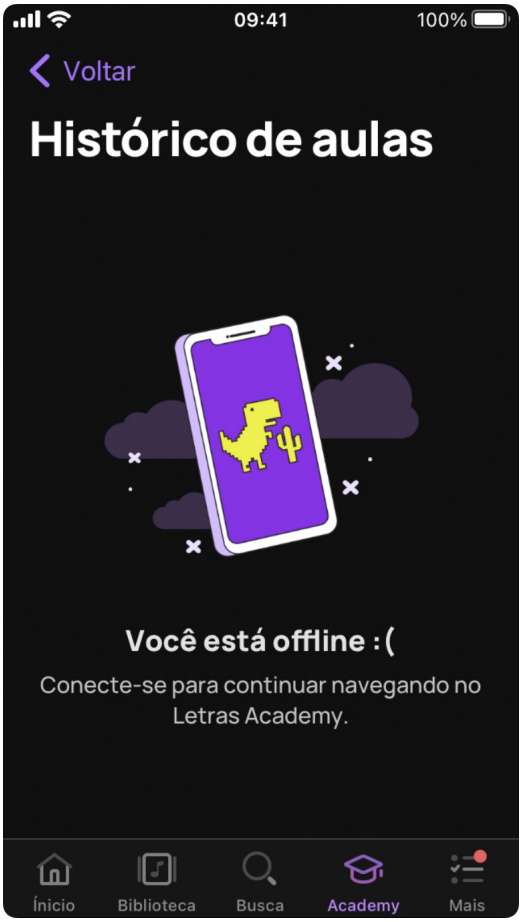


Fonte: Vignoli Comunicação LTDA., 2022

No arquivo da tela de histórico de aulas, foi adicionado um novo componente visual já padronizado de acordo com o *design system* e que representava uma tela de erro. Após a adição deste componente no arquivo *Storyboard*, foi necessário conectá-lo ao arquivo *UIViewControl-*

ler, responsável pela tela, e configurar as regras de restrição de tamanho e espaçamento. Em seguida, com a conexão já efetuada e com as regras configuradas, foi preciso tratar a visibilidade do componente de acordo com o resultado das requisições. Como as requisições desta tela já foram implementadas por outros colegas da equipe, o tratamento da visibilidade da tela de erro se mostrou simples, sendo necessário apenas adicionar uma verificação na resposta da requisição para analisar se houve ou não um erro. A Figura 3.12 apresenta a tela de erro na aplicação em execução.

Figura 3.12 – Tela de erro implementada na atividade para aplicação *offline*



Fonte: Vignoli Comunicação LTDA., 2022

Nesta atividade, a única dificuldade encontrada foi no uso de um objeto que faz monitoramento da conexão de *internet* do dispositivo. Como este objeto era usado em outros pontos no código, tal dificuldade se mostrou ser simples de resolver, tendo sido necessário apenas estudar a implementação feita em outros casos e adaptar para o contexto de erro.

3.2.4 Criação de estrutura para solicitar atualização da aplicação

Por ser um produto que estava sendo desenvolvido à medida em que era pensado, algumas decisões tomadas para o aplicativo *Letras Academy* poderiam não fazer muito sentido no futuro. Além disso, manter um código legado, apenas por retrocompatibilidade de um pequeno número de dispositivos, poderia se tornar inviável. Com isso, se fez necessário criar uma estrutura para forçar a atualização da aplicação *Letras.mus.br*, pois o módulo do *Letras Academy* estava implementado dentro dela.

Tal estrutura foi criada utilizando o mesmo sistema de configuração remota do *Firebase*, cujo funcionamento é descrito na seção 3.1. Para isso, foi criada uma chave cujo valor era um objeto JSON (*JavaScript Object Notation*), o qual descrevia a versão ideal e a versão requerida da aplicação. A versão ideal era utilizada para a exibição de um aviso indicando que havia uma versão mais recente do aplicativo. Já a versão requerida era usada para a exibição de uma tela de erro que impedia o usuário de continuar utilizando o sistema, solicitando que ele efetuasse uma atualização. A Figura 3.13 apresenta um objeto JSON com os dois campos usados: versão ideal e versão requerida.

Figura 3.13 – JSON similar ao utilizado para a funcionalidade de forçar a atualização do aplicativo *Letras Academy*

```
{
    "ideal_version": 261,
    "required_version": 241
}
```

Fonte: Autor

Como o valor da chave era um objeto JSON, foi necessário criar um modelo correspondente em *Swift* e adicionar um *parser* para transformar os dados textuais em um objeto na aplicação. A linguagem *Swift* disponibiliza a classe *JSONDecoder* e o protocolo *Decodable*, que auxiliam nessa transformação. Um exemplo de código que faz a conversão do JSON da Figura 3.13 pode ser visto na Figura 3.14.

Figura 3.14 – Trecho de código para converter um objeto JSON em uma estrutura na linguagem *Swift*

```
struct Versions: Decodable {
    let idealVersion: Int
    let requiredVersion: Int
}

func getVersionsFromJSON(json: String) -> Versions? {
    let decoder = JSONDecoder()
    decoder.keyDecodingStrategy = .convertFromSnakeCase
    return try? decoder.decode(Versions.self, from: Data(json.utf8))
}

let versions = getVersionsFromJSON(
    json: """
          ideal_version: 261,
          required_version: 241
          """
)
```

Fonte: Autor

Apesar de estarem previstos dois casos no escopo da tarefa, no primeiro momento foi implementado apenas o caso onde a atualização era requerida. Assim, foi necessário também de criar a tela exibindo o erro. A tela foi criada dentro de um arquivo do tipo *Storyboard* e, por ter o *layout* parecido com as telas de erro já existentes, a implementação foi simples. A tela criada pode ser vista na Figura 3.15.

Figura 3.15 – Tela para solicitar atualização do *Letras Academy*

Fonte: Vignoli Comunicação LTDA., 2022

Nesta tarefa, as dificuldades encontrada foram no uso da ferramenta de configuração remota do *Firebase*, pois foi implementada em conjunto com a tarefa descrita na seção 3.1, e na implementação da estrutura de transformar um objeto JSON em um objeto da linguagem *Swift*.

3.2.5 Implementação de requisições de pagamento para contratação de professores

Uma das últimas tarefas realizadas durante o estágio foi a implementação de um conjunto de requisições relacionadas à funcionalidade de contratação de professores do *Letras Academy*. Para a implementação das requisições de contratação e pagamento de professores no *Letras Academy*, foi necessário utilizar a biblioteca *Apollo*, responsável por implementar um cliente *GraphQL* para efetuar consultas e mutações. A Figura 3.16 apresenta uma das mutações que foram implementadas na tarefa e que é utilizada pela Vignoli Comunicação LTDA. para a contratação de professores dentro do Letras *Academy*.

Figura 3.16 – Exemplo de mutação implementada para contratação de professores

```
mutation PayContractTeacher(
   $clientMutationID: String!,
   $contractID: ID!,
   $source: String!,
   $paymentMethod: PaymentMethod!,
   $card: PaymentCardInput,
   $billing: PaymentBillingAddressInput,
   $customer: CustomerInput!
) {
   payContractTeacher(input:{
      clientMutationID: $clientMutationID
      contractID: $contractID
      source: $source
      paymentMethod: $paymentMethod
      card: $card
      billing: $billing
      customer: $customer
   }) {
      clientMutationID
      transaction {
         amount
         boletoBarCode
         boletoExpiration
         boletoLink
      }
   }
}
```

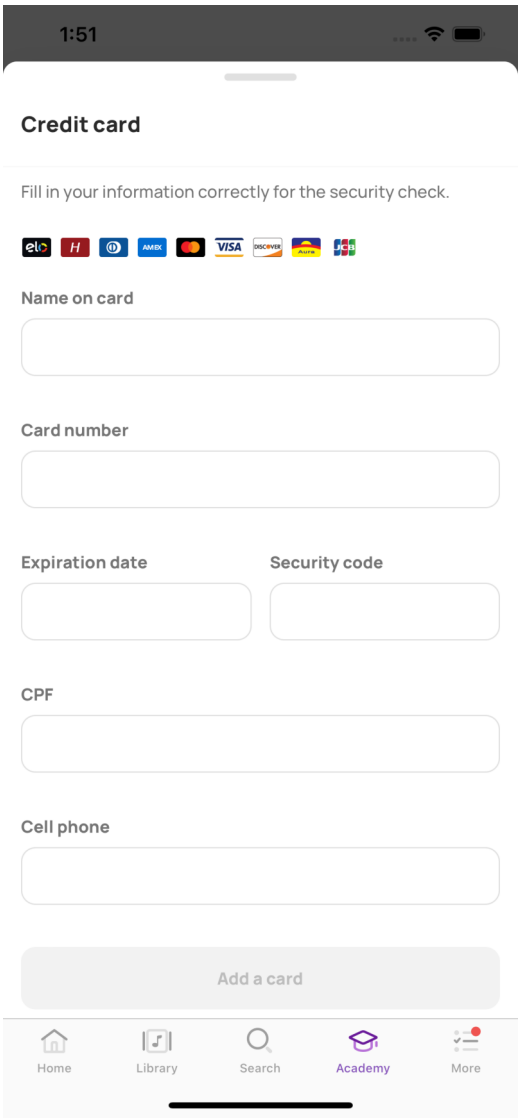
Fonte: Vignoli Comunicação LTDA., 2022

Na mutação apresentada na Figura 3.16, denominada de *PayContractTeacher*, as primeiras oito linhas denominam a mutação e determinam quais são os seus parâmetros. Dentro do escopo da mutação, é feita a chamada da mutação de acordo com a forma em que ela está nomeada no servidor, e nessa chamada os parâmetros da declaração são repassados. Já nas últimas linhas, dentro do escopo da chamada da mutação, são especificados os campos que são retornados, sendo eles um ID do cliente onde a mutação está sendo feita e um objeto do tipo transação, contendo dados como o valor, o *link*, o código de barras e a expiração de um boleto.

Na linguagem *Swift*, a biblioteca *Apollo* facilita a implementação de requisições em *GraphQL*, pois dado um arquivo do tipo *GraphQL* contendo a definição das consultas e mutações, consegue-se gerar automaticamente os modelos definidos e permite o uso destes no código.

Após a implementação das mutações, também foi preciso implementar um controle de estado para o botão de finalizar pagamento, pois alguns dados enviados na mutação são obrigatórios, e integrar as mutações implementadas com as telas de pagamento do *Letras Academy*. Na Figura 3.17 é exibida uma das telas implementadas na aplicação.

Figura 3.17 – Tela de dados de pagamento de professores do *Letras Academy*



Fonte: Vignoli Comunicação LTDA., 2022

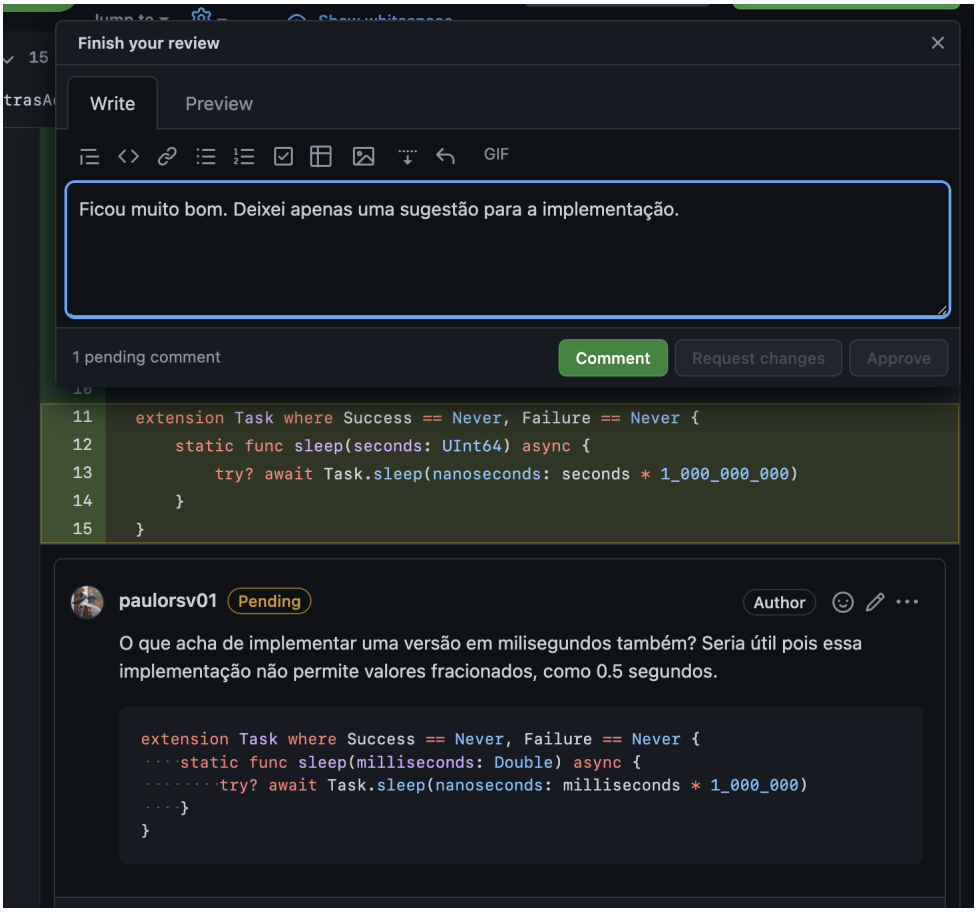
Das dificuldades encontradas, destaca-se o desconhecimento do funcionamento de requisições utilizando *GraphQL* e desconhecimento de como usar a biblioteca *Apollo*. Como outros pontos no código já implementavam mutações, tais dificuldades foram facilmente superadas, sendo necessário apenas estudar como estes outros locais fizeram a implementação e adaptar para o contexto de pagamento.

3.3 *Code review*

Além do desenvolvimento e manutenção de funcionalidades para as aplicações, outra atividade realizada durante o período do estágio foi o *code review*, ou revisão de código. O *code review* é uma prática que consiste na revisão de novas adições e/ou remoções no código-fonte já existente por pessoas diferentes do autor da modificação (MENDES, 2022).

Na Vignoli Comunicação LTDA., todas as modificações feitas nas tarefas passam por este procedimento, com o intuito de interceptar possíveis *bugs* e melhorar o trecho que será incorporado ao código principal. Para isso, é usada a plataforma de controle de versão *GitHub* e sua funcionalidade de *pull requests*. Essa funcionalidade permite que um conjunto de alterações seja acessado, revisado e discutido de maneira simples, permitindo que o processo de *code review* seja realizado. Após o termino de cada tarefa, um novo *pull request* é criado contendo as modificações necessárias para a implementação da mesma. Nestes *pull requests*, é possível adicionar comentários e sugestões em trechos que sofreram modificações e enviar um de três possíveis tipos de *feedback*: aprovar as modificações, dar um *feedback* neutro ou solicitar mudanças. Caso o *feedback* tenha sido de aprovação, a tarefa é mesclada com o código-fonte. Porém, caso o *feedback* tenha sido de solicitar mudanças ou neutro, ele deve ser analisado e posteriormente uma nova revisão é solicitada, recomeçando o ciclo. A Figura 3.18 apresenta a interface de *code reviews* por meio de *pull requests* da plataforma *GitHub*.

Figura 3.18 – Exemplo de *code review* sendo feito por meio do *GitHub*



Fonte: Autor

Durante as atividades de *code review* efetuadas, as maiores dificuldades encontradas foram o desconhecimento das estruturas que já existiam nas aplicações e o desconhecimento da linguagem *Swift*. Para amenizar as dificuldades encontradas nestas atividades, uma solução foi a de ter, no mínimo, duas pessoas efetuando o *code review*.

4 CONSIDERAÇÕES FINAIS

A área da Computação contribui para diferentes setores da sociedade, por possibilitar a solução de problemas e a criação/aprimoramento de produtos. Neste sentido, a realização do estágio na empresa Vignoli Comunicação LTDA. constituiu-se como uma experiência valiosa, que contribuiu para enriquecer a formação acadêmica e fortalecer a articulação entre teoria e prática, no que diz respeito ao desenvolvimento de *software*. A empresa Vignoli Comunicação LTDA. atua na área de entretenimento e educação, dois setores essenciais para a sociedade e o estágio realizado possibilitou o desenvolvimento de competências técnicas e interpessoais necessárias para o profissional da computação.

Durante a realização do estágio, o aperfeiçoamento de competências técnicas ficou evidenciado por meio das atividades realizadas, tais como o desenvolvimento de novas funcionalidades e manutenção de funcionalidades já existentes nos produtos da Vignoli Comunicação LTDA. Isso ocorreu por meio do uso da linguagem *Swift* e suas bibliotecas, de ferramentas de desenvolvimento como a IDE Xcode e o *software Git* e da aplicabilidade dos métodos de desenvolvimento ágil, como *Kanban* e *Scrum*. No período do estágio, devido a equipe de desenvolvedores ter considerado que o *Scrum* implementado apresentava algumas limitações que prejudicavam o fluxo de trabalho, tais como longas reuniões que não eram documentadas e ausência de uma pessoa no papel de *Scrum Master*, optou-se por utilizar a metodologia *Kanban*. A partir da substituição, verificou-se que os problemas existentes foram sanados e que o *Kanban* adequou-se mais ao perfil da equipe.

Em relação às competências interpessoais, as reuniões realizadas, o trabalho em equipe e a busca coletiva por soluções contribuíram para o aprimoramento de *soft skills*, tais como flexibilidade, colaboração, comunicação eficaz, dentre outras, fundamentais para o desenvolvimento profissional.

Algumas disciplinas do curso foram essenciais para um bom aproveitamento do estágio, como por exemplo as disciplinas Introdução aos Algoritmos e Estruturas de Dados. Estas disciplinas desenvolvem o raciocínio lógico do aluno e ensinam conceitos que foram utilizados amplamente no desenvolvimento das tarefas. As disciplinas de Paradigmas de Linguagens de Programação e Práticas de Programação Orientada a Objetos também foram essenciais, pois a linguagem *Swift*, utilizada durante o estágio, possui elementos de múltiplos paradigmas de programação, sendo alguns deles o paradigma funcional e o paradigma orientado a objetos. Por

fim, a disciplina de Engenharia de *Software*, cuja exigência é a entrega de um projeto final, auxiliou no aprendizado de conceitos que puderam ser aplicados na execução do estágio.

A participação em estágios possibilita que o estudante da área da computação vivencie situações-problema em que é preciso aplicar os conhecimentos e boas práticas aprendidas durante o curso antes de terminar a graduação. Isso garante que o estagiário consiga agregar a experiência prática do estágio às disciplinas do curso ainda durante o processo de formação acadêmica. O estágio também abre a possibilidade de que, após a conclusão, o estudante ingresse no mercado de trabalho.

REFERÊNCIAS

APPLE. **Foundation | Apple Developer Documentation**. 2022. Disponível em: <https://developer.apple.com/documentation/foundation>. Acesso em: 07 de set. de 2022.

APPLE. **Swift.org - About Swift**. 2022. Disponível em: <https://www.swift.org/about/>. Acesso em: 07 de set. de 2022.

APPLE. **Xcode 14 Overview**. 2022. Disponível em: <https://developer.apple.com/xcode/>. Acesso em: 07 de set. de 2022.

BYRON, L. **GraphQL: A data query language**. 2015. Disponível em: <https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/>.

CASAGRANDE, E. **Top 100 sites mais acessados no Brasil [Edição 2022]**. 2022. Disponível em: <https://pt.semrush.com/blog/top-100-sites-mais-visitados/>.

CHACON, S.; STRAUB, B. **Pro Git**. 2nd. ed. [S.l.]: Apress, 2014.

FESSENDEN, T. **Design Systems 101**. 2021. Disponível em: <https://www.nngroup.com/articles/design-systems-101>.

MENDES, G. **O que é Kanban? Conheça os principais tipos e como utilizá-los**. 2020. Disponível em: <https://www.fm2s.com.br/tipos-de-kanban/>.

MENDES, R. F. A. **What is Code Review and when should you do it?** 2022. Disponível em: <https://www.imaginarycloud.com/blog/what-is-code-review-and-when-should-you-do-it/>.

SCHWABER, K.; SUTHERLAND, J. **The Scrum Guide - The Definitive Guide to Scrum: The Rules of the Game**. 2020. Disponível em: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pd>.

STEMMLER, K. **GraphQL Mutation vs Query – When to use a GraphQL Mutation**. 2021. Disponível em: <https://www.apollographql.com/blog/graphql/basics/mutation-vs-query-when-to-use-graphql-mutation/>.