Gustavo Storck Andrade de Souza

Desenvolvimento de um aplicativo iOS para gestão de eventos com foco em qualidade

Vitória, ES

2020

Gustavo Storck Andrade de Souza

Desenvolvimento de um aplicativo iOS para gestão de eventos com foco em qualidade

Monografia apresentada ao Curso de Engenharia de Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Engenharia de Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Departamento de Informática

Orientador: Prof. Dr. Vítor E. Silva Souza

Vitória, ES

2020

Gustavo Storck Andrade de Souza

Desenvolvimento de um aplicativo iOS para gestão de eventos com foco em qualidade/ Gustavo Storck Andrade de Souza. – Vitória, ES, 2020-

87 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Vítor E. Silva Souza

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Departamento de Informática, 2020.

1. Swift. 2. Qualidade. I. Souza, Vítor Estêvão Silva. II. Universidade Federal do Espírito Santo. IV. Desenvolvimento de um aplicativo iOS para gestão de eventos com foco em qualidade

CDU 02:141:005.7

Gustavo Storck Andrade de Souza

Desenvolvimento de um aplicativo iOS para gestão de eventos com foco em qualidade

Monografia apresentada ao Curso de Engenharia de Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Engenharia de Computação.

Trabalho aprovado. Vitória, ES, 03 de junho de 2020:

Prof. Dr. Vítor E. Silva Souza
Orientador

Prof[a]. Dr[a]. Monalessa Perini Barcellos
Universidade Federal do Espírito Santo

Camila Zacché de Aguiar
Universidade Federal do Espírito Santo

Vitória, ES
2020

Dedico esse trabalho à minha família, professores e amigos que me apoiaram nessa jornada.

Agradecimentos

Em primeiro lugar agradeço a minha família, em especial a minha mãe e ao meu pai que ao longo desses 22 anos me deram todo suporte e amor necessário além de ensinaram lições que levarei para toda vida. A Viviane Korres meu mais sincero obrigado, por todas experiências, conselhos e risadas que tivemos ao longo desses anos.

Também agradeço todos meus amigos pelos momentos de companheirismo, aprendizado e descontração. Em especial a todos do grupo PET, aos colegas de trabalho do PicPay. E a Gilmarllen Pereira, Felipe Branquinho e Rhuan Caetano por todo apoio nas inúmeras disciplinas.

Gostaria de agradecer também todos professores que me incentivarem a ser cada dia melhor e buscar conhecimento ao longo desses 5 anos de faculdade. Em especial ao meu orientador Vítor e às professoras Roberta e Patrícia, tutoras do PET na época em que fui bolsista.

Por fim, agradeço a Deus por ter me guiado e me permitido conhecer pessoas maravilhosas.

"The only way to go fast, is to go well."
(Robert Cecil Martin)

Resumo

Hoje, diante da grande transformação cultural e tecnológica das últimas décadas, é cada vez mais comum encontrarmos serviços, entretenimento e produtos baseados em softwares. Com essa grande evolução, os projetos têm um escopo cada vez mais dinâmico e são desenvolvidos por equipes cada vez maiores. O mercado exige que as aplicações sejam entregues para o usuário de forma confiável demandando o menor tempo e custo possível.

Diante desse cenário, o desafio deste trabalho é estudar métodos, formas de desenvolvimento que garantam a escalabilidade, testabilidade e manutenibilidade. Iremos aplicar uma esteira de desenvolvimento que ajude a manter essas características e facilite o desenvolvimento da aplicação, usando conceitos de CI/CD, padrão de código. Neste trabalho, discutimos vários pontos relacionados a este contexto, desde o processo de gerência de software e prototipação, até a escolha de uma arquitetura e implementação de testes. Todos esses conceitos serão utilizados no desenvolvimento de um aplicativo *mobile* iOS nativo, para gerência de eventos, por meio do qual o usuário pode interagir e se informa sobre as atividades e notícias. Esse aplicativo deve ser customizável para cada cliente e possuir uma mesma base de código, de modo que a adição e configuração de um novo evento sejam simples e rápidas.

Palavras-chaves: Desenvolvimento iOS, Qualidade, Teste, Swift, CI, CD.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

UML	Unified Modeling Language
GCD	Grand Central Dispatch
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
MVVM-C	Model-View-ViewModel-Coordinator
VIPER	View-Interactor-Presenter-Entity-Router
VIP	View-Interactor-Presenter
CI	Continuous Integration
CD	Continuous Delivery
IDE	Integrated Development Environment
XIB	XML Interface Builder
SDK	Software Development Kit

Sumário

1 Introdução

Vivemos em mundo cada vez mais conectado: estima-se que 56,1% da população mundial tem acesso à Internet. Esse processo desencadeou mudanças profundas na sociedade. Em 1833 o serviço postal de Paris a Strasburgo levava 36 horas, a notícia da queda da bastilha chegou a Madri em 13 dias (HOBSBAWN, 2012, p. 25). Hoje podemos enviar uma mensagem para qualquer lugar do mundo em uma fração de segundos. Nossa forma de se comunicar, fazer compras, buscar informação, administrar nosso dinheiro foi totalmente alterada em menos de um século.

No meio desse processo de evolução e transformação, um dos mecanismos fundamentais é o software, pois ele define como um determinado problema será efetivamente resolvido, fazendo a interface entre a demanda do usuário e o poder computacional para obter uma solução. Um dos principais desafios na elaboração de um software é desenvolver com qualidade, não bastando focarmos apenas no resultado do problema (MARTIN, 2009). Certamente é importante alcançarmos um objetivo, seja ele fazer um cálculo ou realizarmos um pagamento a um amigo, mas também devemos dar valor ao processo de resolução do problema. Um software bem desenvolvido garante estabilidade, escalabilidade, fácil manutenção. Quando esses fatores são atingidos conseguimos uma melhor experiência do usuário, economia de dinheiro e tempo (MARTIN, 2018). Este trabalho propõe estudar como podemos construir esse tipo software e então aplicar métodos adequados no desenvolvimento de um aplicativo, que visem escalabilidade, testabilidade e manutenibilidade para conseguirmos atingir a qualidade.

1.1 Contexto

O objetivo deste projeto será desenvolver um aplicativo iOS para a Eventu, um sistema para organização e gerência de eventos capaz de criar e personalizar um site e aplicativos nativos (Android e iOS) para apoio à realização de um evento (mais detalhes sobre os requisitos da Eventu são apresentados no Capítulo 4).

Para o estudo e aplicação de métodos que garantam a qualidade, nos baseamos no processo de software descrito na Figura 1. Todas as partes foram contempladas, mas nosso maior foco foi na região destacada na figura. Aplicamos métodos ágeis e modernos que visem desburocratizar o processo e atingir o objetivo.

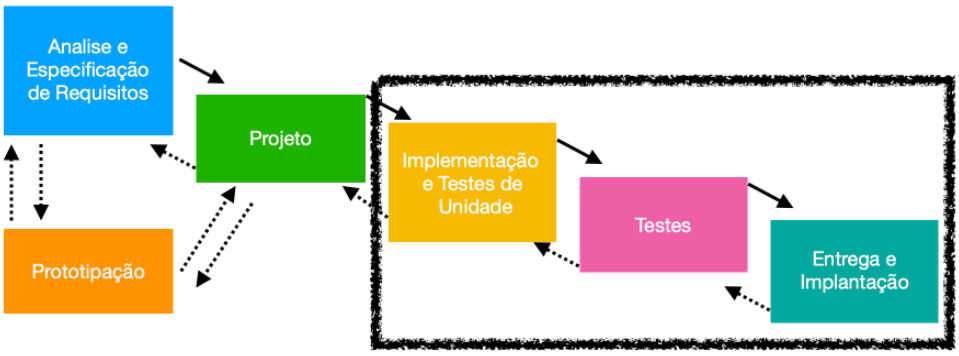

Figura 1 – Processo de software seguido neste trabalho.

1.2 Motivação e Justificativa

Em novembro de 2017, eu e meu sócio fundamos a Eventu, a demanda de mercado era clara, a maioria dos softwares para gerência de evento eram muito ruins e incompletos ou muito caros, suprindo mais que o necessário às demandas da organização. A ideia surgiu quando eu participava do PET (Programa de Ensino Tutorial) e trilhava meus primeiros passos no desenvolvimento de aplicativos móveis para Android. Para validar os conhecimentos aprendidos, estava construindo, em conjunto com a equipe, um aplicativo para auxiliar participantes do Enapet (Encontro Nacional dos Grupos do Programa de Educação Tutorial) e foi uma experiência muito enriquecedora.

Após o evento a tutora do grupo enxergou um grande potencial e me incentivou a empreender com essa ideia. Alguns meses depois, em 2018, após muito trabalho meu e de meu sócio, ele responsável pelo site e *backend* e eu pelo aplicativo Android, concluímos a primeira versão do sistema. Nesse mesmo ano conseguimos dois clientes e seus eventos foram um sucesso. Porém, problemas aconteceram: o código era extremamente difícil de dar manutenção e adicionar novas funcionalidades; alguns *bugs* eram relatados mas não tínhamos como simular ou saber em qual situação aconteceram; ao alterar parte do código acabávamos inserindo problemas em outras partes que o reutilizavam; o processo de versionamento e publicação era sem controle; dentre outros. Estava claro que algo tinha que mudar.

No ano de 2018 também tive a oportunidade de trabalhar na PicPay,[1] um aplicativo de pagamentos usados por milhões de pessoas. Essa experiência alterou completamente a minha forma de pensar em desenvolvimento de software. Foi o início da minha busca por processos e métodos de desenvolvimento capazes de assegurar qualidade do produto (ROCHA; MALDONADO; WEBER, 2001).

Qualidade é um dos fatores primordiais para se garantir a evolução de um produto. É comum ouvirmos histórias de empresas que aumentaram o número de desenvolvedores e

[1] <https://picpay.com/>

a média de código entregue diminui. Isso ocorre devido ao estado do código, da arquitetura e, de uma forma geral, aos processos de desenvolvimento. De acordo com Duarte e Falbo (2000), a qualidade do produto depende fortemente da qualidade de seu processo de desenvolvimento. A falta de qualidade acontece pois muitas vezes colocamos velocidade de entrega em detrimento da qualidade e ficamos satisfeitos em entregar um projeto no prazo, sendo o maior objetivo apenas funcionar. Há um pensamento que traduz muito bem essa ideia:

> *If you give me a program that works perfectly but is impossible to change, then it won't work when the requirements change, and I won't be able to make it work. Therefore the program becomes useless.*
>
> *If you give me a program that does not work but is easy to change, then I can make it work, and keep working as requirements change. Therefore the program will remain continually useful. (MARTIN, 2018, p. 15)*

Parte da culpa é nossa, como programadores, por não projetarmos e codificarmos da maneira correta. Essa foi a maior motivação do projeto: produzir um software cujo resultado final agregue valor ao cliente e aos desenvolvedores, fazendo um *trade-off* entre qualidade e tempo.

1.3 Objetivos

O objetivo geral deste trabalho é desenvolver um aplicativo iOS com o foco em desenvolvimento com qualidade, a ser atingido por meio da implementação de testes, módulos, padrões de código, aplicando uma esteira de desenvolvimento que implemente integração e entrega contínuas e tomando decisões arquiteturais que tornem o código mais escalável, reusável, isolado e de fácil manutenção.

São objetivos específicos do trabalho:

- Documentar os requisitos conforme os objetivos solicitados pela Eventu;
- Prototipar uma interface que atenda a demanda do projeto e as *guidelines* da Apple para aplicativos iOS;
- Documentar o projeto de software, definindo sua arquitetura;
- Organizar um *pipeline* de desenvolvimento, que possa ser automatizado com o desenvolvimento;
- Criar uma biblioteca privada que reúna os componentes gráficos que foram criados especificamente para Eventu;

- Criar uma biblioteca pública de código aberto, para um componente gráfico que se adeque a outros projetos, usando como canal de distribuição o *CocoaPods*;[2]
- Implementar testes unitários e de interface;
- Organizar uma implantação que seja gradual e possua versionamento da base;
- Usar essa solução no Congresso Brasileiro de Engenharia Agrícola(CONBEA), que ocorrerá na segundo semestre de 2019, de modo a coletar informações de uso e problemas no aplicativo, avaliando o resultado do projeto.

1.4 Método de Desenvolvimento do Trabalho

No início do trabalho foram realizadas reuniões de requisitos e prototipação, para modelagem do sistema. As análises de requisitos tiveram como base os fundamentos teóricos da disciplina de Engenharia de Software. Após termos os requisitos iniciou-se o processo de prototipação usando como referencia o conteúdo da disciplina optativa Interface Humano Computador, além do estudo da *Interface Guidelines*[3] da Apple, usando o Figma[4] como software de desenho das telas. Os artefatos gerados por esses dois processos foram utilizados como descrição das tarefas, cadastradas no GitProject,[5] para acompanhamento do projeto, esses artefatos são apresentados no Capítulo 4.

Para definir e organizar um pipeline de desenvolvimento foi utilizado como base o conhecimento adquirido em participação de eventos focados no assunto, estudos de casos de grandes empresas de tecnologia, *podcasts* e leitura de artigos. As ferramentas auxiliares nesse passo foram definidas através de pesquisas e testes das existentes no mercado, levantando custo vs. benefício para sua implantação, com apoio de suas documentações. As ferramentas e suas implementações estão disponíveis no Capítulo 3.

Para temas amplos que exigem conhecimento vasto, como definição da arquitetura, linguagem de programação Swift,[6] desenvolvimento iOS e testes, me baseei em livros e artigos que discutem as escolhas levando em consideração a qualidade como argumento. No desenvolvimento do projeto foram utilizadas bibliotecas de terceiros que auxiliaram a atingir alguns objetivos como organizar uma implantação que seja gradual e possua versionamento da base implementado a partir do *Remote Config*[7] do *Firebase*.[8] Algumas dessas bibliotecas que eram de código aberto serviram de referencial para implementarmos

[2] <https://cocoapods.org>
[3] <https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/>
[4] <https://www.figma.com/>
[5] <https://github.com/features/project-management/>
[6] <https://swift.org/documentation/>
[7] <https://firebase.google.com/docs/remote-config>
[8] <https://firebase.google.com>

as nossas próprias bibliotecas públicas e privadas e fazer a distribuição no gerenciador de dependência CocoaPods.[9] Também nos baseamos em padrões de projetos e código para construção do projeto, todos esses métodos em conjunto com a documentação da arquitetura podem ser vistos no Capítulo 5.

Como resultado aplicamos todos os conceitos apresentados no trabalho para desenvolvimento de um aplicativo iOS, usado no Congresso Brasileiro de Engenharia Agrícola (CONBEA) 2019.

Pretende-se mostrar, com este trabalho, que a graduação forneceu a base sólida, para que eu continuasse a aprender, evoluir e analisar com criticidade novas ideias propostas.

1.5 Organização do Texto

Esta monografia, entregue como produto final deste trabalho, é dividida da seguinte maneira:

- Capítulo 2 – Fundamentação Teórica e Tecnologias Utilizadas: apresenta uma discussão sobre tecnologias para desenvolvimento *mobile* e introduz as principais ferramentas que foram utilizadas no decorrer do projeto;
- Capítulo 3 – Apoio ao Desenvolvimento: apresenta todas as ferramentas e métodos que foram utilizados para garantir ou auxiliar um desenvolvimento com qualidade e escalabilidade;
- Capítulo 4 – Especificação de Requisitos: apresenta o diagrama de classes UML, a tabela de requisitos e o protótipo da interface;
- Capítulo 5 – Projeto e Implementação: descreve a implementação das principais tecnologias e conceitos utilizados neste trabalho e apresenta todo o sistema desenvolvido, descrevendo os fluxos e comportamentos, ilustrados por capturas de tela;
- Capítulo 6 – Considerações Finais: apresenta as conclusões do trabalho, dificuldades e ensinamentos que o projeto proporcionou;

[9] <https://cocoapods.org>

2 Fundamentação Teórica e Tecnologias Utilizadas

Neste capítulo analisamos os métodos e tecnologias usadas como base para realização deste trabalho. A Seção 2.1 analisa as vantagens e desvantagens das diferentes abordagens para desenvolvimento de aplicativos *mobile*, justificando a escolha da abordagem utilizada neste trabalho; a Seção 2.2 relata tarefa análoga, porém em relação às diferentes arquiteturas utilizadas em projetos para esta plataforma; a Seção 2.3 discute sobre a importância e os principais tipos de testes; a Seção 2.4 apresenta os conceitos de CI e CD; a Seção 2.5 discorre sobre as 3 principais formas de se implementar interfaces gráficas no iOS; a Seção 2.6 introduz os conceitos de *Clean Code* e *Clean Architecture*; a Seção 2.7 apresenta o *Scrum*; e, finalmente, a Seção 2.8 menciona as ferramentas de apoio para realização deste fluxo de trabalho.

2.1 Abordagens

Um requisito fundamental para Eventu é que o aplicativo execute na plataforma iOS. iOS é o sistema operacional *mobile* usado pelos iPhones, desenvolvido pela Apple e apresentado no ano de 2007. A empresa adota uma medida protecionista de mercado onde ela só permite que seus dispositivos *mobile* executem o sistema operacional próprio. Para desenvolver um aplicativo para iPhone é necessário que ele seja desenvolvido para iOS e distribuído pela AppStore.[1]

Hoje no mercado existem três principais abordagens para desenvolvimento de software para esta plataforma: Nativo, Híbrido e *Cross Platform*. Esta seção destina-se a apresentar cada uma das alternativas, mostrando seus pontos fortes e fracos. A escolha da plataforma é essencial pois impacta no desenvolvimento e resultado do produto.

2.1.1 Nativo

São aplicativos desenvolvidos para uma plataforma específica, nas linguagens aceitas por essa plataforma. Por esse motivo acabam tendo acesso direto às *frameworks* nativas que fazem a interface com o sistema operacional. Para o iOS temos uma arquitetura em camadas, ilustrada na Figura 2.

[1] <https://www.apple.com/ios/app-store/>

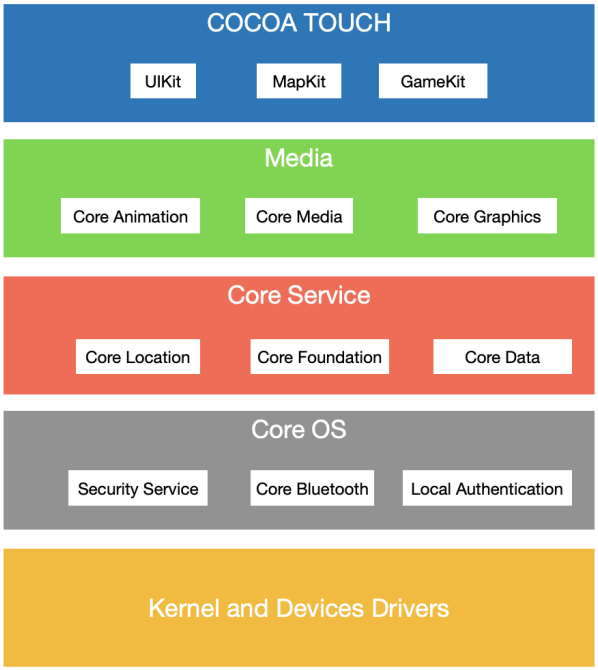

Figura 2 – Arquitetura em camadas iOS[2]

Cocoa Touch: é a principal responsável pela aparência do aplicativo e sua capacidade de resposta às ações do usuário, fornecendo a infraestrutura básica para um conjunto de tarefas chave como multitarefa e *Auto Layout*. A principal *framework* dessa camada é a *UIKit*, responsável pelo suporte a aplicações gráficas orientadas a eventos. Ela fornece a janela e a arquitetura de visualização para implementar a interface e a infraestrutura de manipulação de eventos.

Media: fornece os recursos de áudio, vídeo, animação e gráficos. Assim como acontece com as outras camadas, a camada de mídia compreende uma série de estruturas que podem ser utilizadas no desenvolvimento de aplicativos para iPhone.

Core Service: fornece grande parte da base sobre a qual as camadas acima são construídas. Ele fornece armazenamento no iCloud, internacionalização, localização, proteção de dados, suporte a compartilhamento de arquivos, GCD (API para trabalhar com *threads*).

Core OS: essa camada é responsável pelo gerenciamento da memória, alocando e liberando memória, cuidando das tarefas do sistema de arquivos, do gerenciamento da rede e de outras tarefas do sistema operacional. Também interage diretamente com o hardware.

Prós:

[2] <https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html>

- Fornece todos os recursos disponíveis no dispositivo e sistema operacional;
- Alto desempenho e velocidade;
- Usa tecnologias recomendadas e apoiadas pelos desenvolvedores do sistema operacional, no caso do iOS;
- Oferece experiência de usuário compatível com o sistema operacional;
- Segurança;
- Desenvolvimento maduro e consolidado, que já está sendo evoluído e testado há anos, com grande apoio da comunidade e extensa documentação.

Contras:

- Os códigos não são compartilhados entre as diferentes plataformas de dispositivos móveis, logo é necessário manter um time de desenvolvedores para cada plataforma afetando o preço e tempo de desenvolvimento.

2.1.2 Híbrido

Tecnologias híbridas permitem usar desenvolvimento Web (HTML, CSS, JavaScript) para criar aplicativos que executem em múltiplas plataformas (Android, iOS, *desktop*, navegadores), usando a mesma base de código. Elas possuem bibliotecas que simulam elementos visuais e comportamentos nativos, são executadas dentro de contêineres, no *Mobile Web Views*, por onde pode ser feita a integração com *hardware*, como abrir a câmera. É muito similar a um site na Web, com a diferença que pode ser obtido e executado como um aplicativo no dispositivo móvel. Ferramentas como o Ionic[3] e o Apache Cordova[4] permitem o desenvolvimento de aplicações utilizando esta abordagem.

Prós:

- Utiliza a mesma base de código, reduzindo as equipes e tempo de desenvolvimento;
- Portabilidade, pode ser executado em múltiplas plataformas, iOS, Android, navegadores e *desktops*.

Contras:

- Baixo desempenho;
- Experiência do usuário incompleta;

[3] <https://ionicframework.com>
[4] <https://cordova.apache.org>

- Alta dependência de bibliotecas;
- Dificuldade para integrar com funcionalidades do sistema operacional, como *push notification*, por exemplo.

2.1.3 *Cross Platform*

Utiliza a mesma base de código para criar aplicativos que executem em múltiplas plataformas (ex.: Android, iOS). A maior diferença em relação aos aplicativos híbridos é que o *Cross Platform* utiliza bibliotecas que traduzem o seu código em elementos visuais e comportamentos nativos de cada plataforma, além do aplicativo ser também executado de forma nativa, o que traz um grande ganho de desempenho e usabilidade em relação ao híbrido. Ferramentas como o React Native,[5] o Xamarin[6] e o Flutter[7] permitem o desenvolvimento de aplicações utilizando esta abordagem.

Prós:

- Utiliza a mesma base de código, reduzindo as equipes e tempo de desenvolvimento;
- Portabilidade, pode ser executado em múltiplas plataformas como, por exemplo, iOS e Android.

Contras:

- Médio desempenho;
- Experiência do usuário incompleta;
- Personalização dependente da *framework*;
- Pela grande dependência de bibliotecas, o *build* final fica maior que o nativo.

2.1.4 Conclusões

Nosso projeto foi desenvolvido de maneira nativa, por oferecer a melhor experiência do usuário com base na velocidade, conformidade da plataforma e uso dos recursos mais recentes. A Apple continua investindo na abordagem nativa, criando novas tecnologias, como o SwiftUI[8] e melhorando as existentes, como Xcode[9] e Swift[10], otimizando a compilação, melhorando tempo de execução, adicionando mais funcionalidades às linguagens, com excelente suporte, documentação e envolvimento da comunidade.

[5] <https://facebook.github.io/react-native/>
[6] <https://visualstudio.microsoft.com/xamarin>
[7] <https://flutter.dev>
[8] <https://developer.apple.com/xcode/swiftui/>
[9] <https://developer.apple.com/xcode/>
[10] <https://swift.org>

2.1.5 Linguagem de Programação

Para o desenvolvimento nativo na plataforma iOS, pode-se utilizar como linguagem de programação Objective-C ou Swift (LECHETA, 2017).

O Objective-C foi criado no início da década de 1980 por Brad Cox e Tom Love como uma extensão do C. Fornece recursos orientados a objetos e um tempo de execução dinâmico. O Objective-C herda a sintaxe, os tipos primitivos e as instruções de controle de fluxo de C e adiciona a sintaxe para definir classes e métodos. No entanto, atualmente é uma linguagem com características ultrapassadas, por exemplo: ainda existem ponteiros explícitos, alocação de memória, arquivo de cabeçalho (`.h`), código muito verboso, não apresenta elementos funcionais, não possui *Null Safety*.

Observando a defasagem e os crescentes problemas com a linguagem, a Apple em 2014 resolveu atender a comunidade e lançou o Swift, uma linguagem orientada a objetos, estruturada, imperativa, compilada, concorrente e funcional. Além disso, a linguagem ainda mantém compatibilidade com código existente em Objective-C.

Apesar da Apple dar suporte às duas linguagens, para este projeto adotamos o Swift, por ser mais moderna, segura e fácil de aprender.

2.2 Arquiteturas

Nessa seção discutimos os principais padrões de arquiteturas utilizadas no desenvolvimento de aplicativos *mobile*. A escolha de uma arquitetura correta implica em melhor isolamento do código, melhor escalabilidade, facilitando a manutenção e aplicação de testes.

2.2.1 MVC (*Model-View-Controller*)

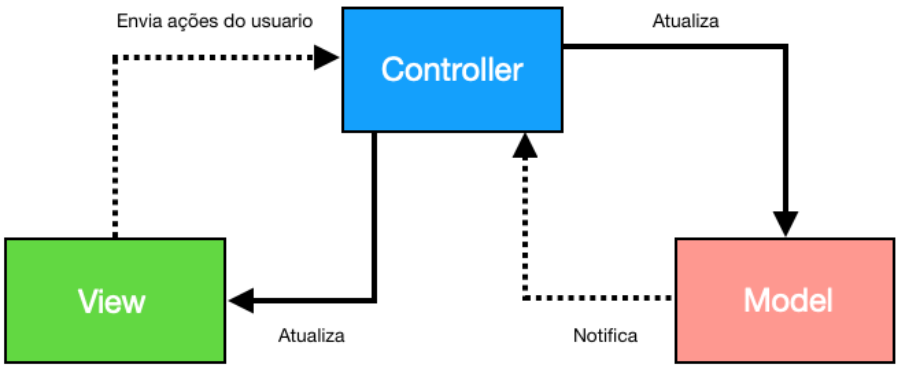

Figura 3 – Arquitetura MVC

O padrão MVC[11] (*Model-View-Controller*), ilustrado na Figura 3, é o padrão mais simples e foi durante muitos anos o recomendado pela Apple. Nessa arquitetura, a *View* é responsável pela apresentação da interface, e notifica o *Controller* sobre as ações do usuário como, por exemplo, o clique de um botão. O *Model* representa os dados do seu domínio, por exemplo uma classe Filme. O *Controller* tem a função de atualizar a *View* e o *Model*, garantindo a comunicação entre os dois. A *View* e o *Model* não conhecem o *Controller*, elas notificam suas mudanças de estados e esperam alguém reagir a isso.

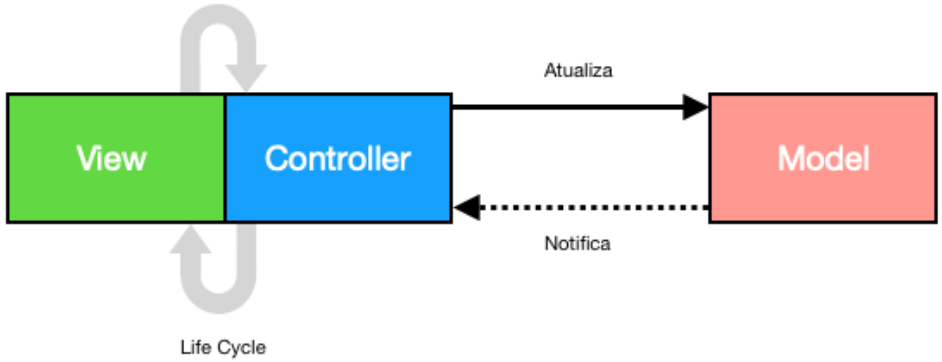

Figura 4 – Arquitetura MVC na realidade.

Vale destacar que o MVC no desenvolvimento *mobile* é adaptado às características desta plataforma, portanto se distingue do MVC tradicional ilustrado anteriormente. Uma representação mais fiel dessa arquitetura no desenvolvimento iOS seria a Figura 4,[12] na qual percebemos como *View* e *Controller* são fortemente conectados pelo ciclo de vida. Em um padrão MVC isso é um problema, pois força o controlador a lidar com inúmeras responsabilidades. Os principais problemas dessa abordagem é a dificuldade de realizar testes unitários e a precária divisão de responsabilidade, ferindo os princípios da responsabilidade única.

[11] <https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>

[12] <https://engineering.etermax.com/dealing-with-massive-view-models-using-mvvm-on-ios-74b2697557ce>

2.2.2 MVP (*Model-View-Presenter*)

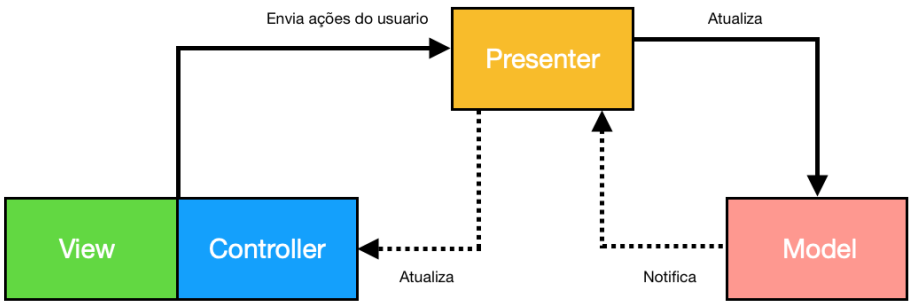

Figura 5 – Arquitetura MVP.

O MVP[13] (*Model-View-Presenter*), ilustrado na Figura 5, é uma extensão do MVC que trata o *Controller* como parte da *View*, distribuindo sua responsabilidade com o *Presenter*. A *View* continua com a função de lidar com a interface do usuário e o *Model* com o domínio, enquanto o *Presenter* trata da lógica de negócios e apresentação. Dessa forma fica mais fácil de aplicar testes unitários, pois toda lógica de negócio está isolada e não depende do ciclo de vida da *View*. Em relação ao MVC, ele distribui melhor as responsabilidades, o que diminui o custo de manutenção e melhora a escalabilidade.

2.2.3 MVVM (*Model-View-ViewModel*)

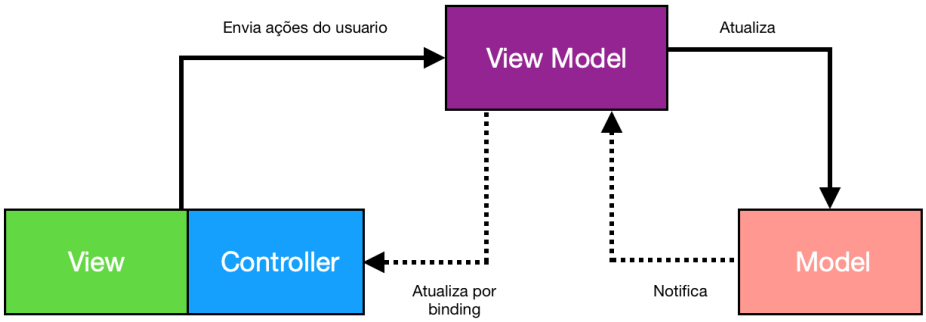

Figura 6 – Arquitetura MVVM

O MVVM[14] (*Model-View-ViewModel*), ilustrado na Figura 6, foi apresentado pela Microsoft em 2005 para facilitar a programação orientada a eventos e desde então vem sendo amplamente adotado no desenvolvimento *mobile*. O *Model* e a *View* continuam sendo os mesmos que no MVP, mas aqui o intermediário *View Model*, atualiza a *View*

[13] <https://medium.com/@saad.eloulladi/ios-swift-mvp-architecture-pattern-a2b0c2d310a3>
[14] <https://www.appcoda.com/mvvm-vs-mvc/>

usando *data binding*. Ao contrário do *Presenter*, que realiza chamadas diretas a *View*, o *View Model*, expõe eventos de mudança que são utilizados pelo *Controller* para alterar a interface.

2.2.4 MVVM-C (*Model-View-ViewModel-Coordinator*)

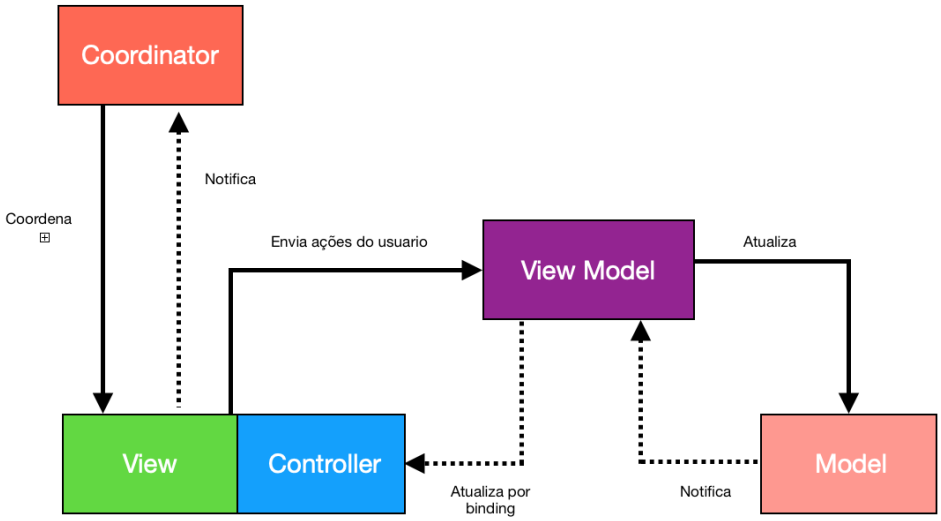

Figura 7 – Arquitetura MVVM-C.

O MVVM-C[15] (*Model-View-ViewModel-Coordinator*), ilustrado na Figura 7, introduz um novo componente ao MVVM tradicional, o *Coordinator*, que nos permite fazer o controle do fluxo das telas. Além disso, ele torna o *Controller* ainda mais passivo tirando a responsabilidade de orquestrar o fluxo das telas e fazer a injeção de dependência. Até então nas arquiteturas anteriores era dever do *Controller* instanciar, adicionar e retirar da pilha de navegação outro *Controller*, o que cria dois problemas. Primeiro, em uma arquitetura que preza o isolamento uma *View* não deveria saber em qual parte do fluxo está inserida e muito menos as regras de negócio para dar continuidade. O *Controller* deve se preocupar somente em responder às ações da sua interface. O segundo problema que o *Coordinator* resolve é a comunicação entre as telas. Imagine o seguinte caso: temos algumas telas na nossa pilha de navegação que representam um fluxo de login. Ao final do login temos que notificar a segunda tela da pilha que tudo foi realizado com sucesso. Nesse caso o último *Controller* não conhece a segunda tela e sem alguém responsável por coordenar o fluxo teríamos um problema. O MVVM-C usa as vantagens conhecidas do MVVM e fornece uma solução para o problema de rotas no desenvolvimento *mobile*.

[15] <https://medium.com/sudo-by-icalia-labs/ios-architecture-mvvm-c-introduction-1-6-815204248518/>

2.2.5 VIPER (*View-Interactor-Presenter-Entity-Router*)

O VIPER[16] (*View-Interactor-Presenter-Entity-Router*), ilustrado na Figura 8, eleva a divisão de responsabilidades e também se preocupa com o roteamento das telas, fazendo um fluxo bidirecional dos dados. Nesta arquitetura existem cinco camadas:

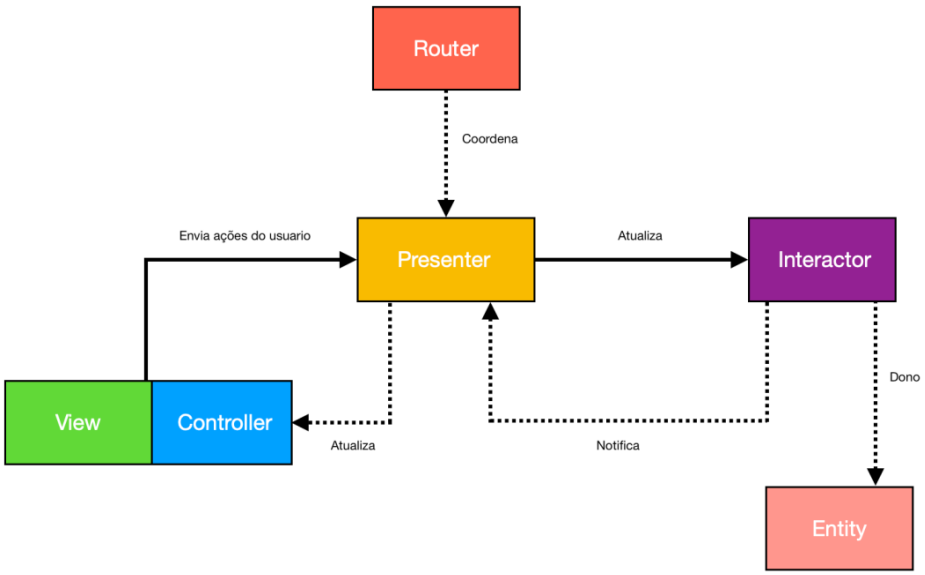

Figura 8 – Arquitetura Viper.

- ***Interactor***: contém a lógica de negócio relacionada aos dados,
- ***Presenter***: contém a lógica de negócios da UI, chama métodos do *Interactor*;
- ***Entity***: objetos de domínio, são usados apenas para representar e armazenar, todo tratamento de dados é feito pelo *Interactor*;
- ***Router***: responsável pela navegação entre telas, muito parecido com o *coordinator* do MVVM-C;
- ***View***: continua a mesma das outras arquiteturas, sendo compostas pela *View* e pelo *Controller*.

O VIPER se destaca com uma proposta de arquitetura muito bem definida, onde cada camada possui uma única responsabilidade.

[16] <https://www.raywenderlich.com/8440907-getting-started-with-the-viper-architecture-pattern>

2.2.6 Clean Swift ou VIP (*View-Interactor-Presenter*)

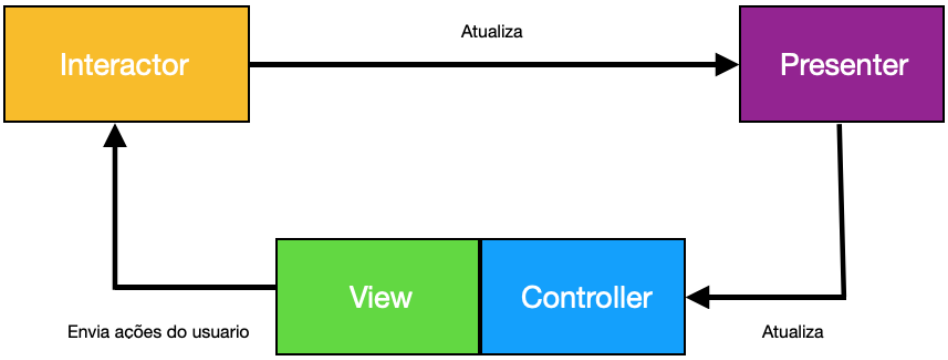

Figura 9 – Arquitetura Vip.

O VIP[17] (*View-Interactor-Presenter*), ilustrado na Figura 9, segue a mesma divisão de responsabilidades do VIPER, mas define menos atores na arquitetura. Sua principal vantagem quanto ao VIPER é que ele apresenta um fluxo unidirecional de dados.

2.3 Testes

Testes são importantes para evitar que mudanças alterem comportamentos definidos anteriormente de forma inesperada. A implementação dos testes é feita em conjunto com a sua tarefa, logo uma tarefa compreende resolver um determinado problema e escrever testes para essa solução.

Um dos conceitos mais bem difundidos sobre teste é a Pirâmide de Testes,[18] que divide os testes em 3 categorias, conforme mostra Figura 10:

[17] <https://clean-swift.com/>
[18] <https://medium.com/pacroy/separate-unit-integration-and-functional-tests-for-continuous-delivery-f4dc240d8f2f>

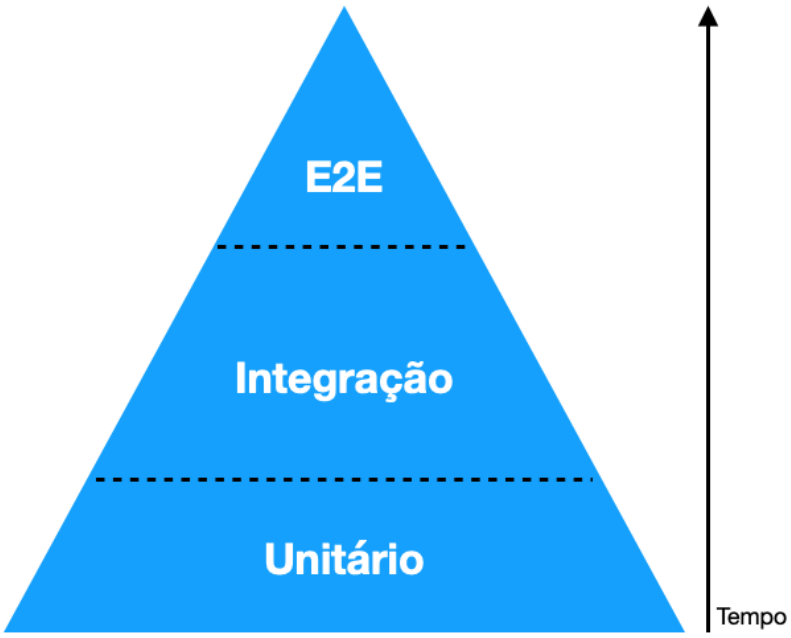

Figura 10 – Pirâmide de Teste

- **Unitário**: busca testar a menor unidade que temos no código a função. Ele analisa a saída para um conjunto de entradas e verifica se o resultado é o esperado. Garante, de forma automatizada, que a função não tenha seu comportamento alterado de forma acidental;
- **Integração**: testam o funcionamento da união de componentes, podem existir situações em que os teste unitário validam 2 componentes de código como corretas, mas a integração entre as duas não surtir o resultado esperado. A integração testa o resultado da união desses componentes;
- **E2E**: teste de ponta a ponta ou funcionais, procuram garantir que toda a aplicação, com suas funções e integrações, funciona como esperado. Para programação *mobile* é muito comum utilizarmos teste de interface para fazermos essa verificação.

É interessante observar que ao subir na pirâmide os testes ficam computacionalmente mais caros de serem realizados, demandando mais tempo para serem executados, ao mesmo tempo o topo da pirâmide aproxima os testes para uma forma mais real da interação do usuário com aplicação. É um *trade-off* entre tempo de desenvolvimento e veracidade do cenário de teste.

2.4 CI e CD

CI (*Continuous Integration*) e CD (*Continuous Delivery*) são dois acrônimos que costumam ser mencionados quando as pessoas falam sobre práticas de desenvolvimento

modernas. CI significa integração contínua: para cada tarefa concluída as alterações são validadas criando um *build*, e executando os testes automatizados. Na nossa integração, uma tarefa só é considerada concluída caso esse processo seja feito com sucesso. CD significar entrega contínua, ele automatiza o processo de publicação do aplicativo na *AppStore* e no *TestFlight*, por meio dele podemos definir regras, como, em quantos tempo x% da base receberá a atualização, quem tem acesso ao aplicativo em fase beta, etc.

2.5 Views

Na programação para dispositivos iOS existem 2 formas de se construir uma interface: a primeira via *interface builder* e a segunda via código, também conhecida com *View Code*. Essa seção irá introduzir as duas formas.

2.5.1 Interface Builder

Interface builder é método visual de se implementar *views*, ele consiste em selecionar os elementos e organizá-los de forma a compor uma interface. Eles são organizados através de *constraints*, que definem ordem e prioridade por meio de âncoras: cada elemento deve estar ancorado em relação ao elemento pai ou a um elemento adjacente. É um método bem rápido e intuitivo de desenhar interfaces, que utiliza recurso de arrastar e soltar objetos, permitindo configurá-los por meio de uma janela de especificações. É possível fazer referência no código aos elementos gráficos por links simbólicos, chamados de *IBoutlets*. Podem ser implementados de duas formas:

- ***Xib***: representa uma tela completa, ou um componente que pode ser reutilizado por outros *Xibs*;
- ***Storyboards***: é um conjunto de *Xibs*, tem o poder de representar o fluxo e fazer a navegação entre as telas.

Toda *view* gerada via *Interface Builder* é transformada em um arquivo *XML*, representando toda hierarquia e atributos dos componentes. O grande problema é que, diferente de outros plataformas, como o Android, esse *XML* não tem a menor intenção de ser compreendido por seres humanos, o que acaba afetando drasticamente o *code review*, pois não é possível apontar erros ou melhorias. Quando existe conflito de *merge*, isso se agrava tendo que excluir uma das partes e refazê-la depois, sobre o novo código adicionado. A criação de uma *ViewController* é baseada em um *identifier*, uma string, que é definida no *xib* e, caso esse identificador não seja encontrado ou um *IBoutlets* esteja incorreto, a instância falha e o aplicativo é terminado (*crash*), em tempo de execução. Por ser muito suscetível a erro humano e não ser verificada no processo de compilação o *interface builder*

acaba se tornando um método inseguro para times e aplicativos grandes.

O *Storyboard* agrava o problema do conflito: por representar um fluxo, existe mais probabilidade de existirem conflitos. Outro grande problema é a injeção de dependência: quando construímos uma *ViewController* baseada no *interface builder*, perdemos a capacidade de usar o seu construtor para passagem de parâmetros, sendo necessário fazer a injeção indireta de dependência por variáveis públicas e opcionais.

2.5.2 *View Code*

View Code é a implementação de *Views* utilizando a mesma linguagem de programação escolhida para o desenvolvimento do *App*. Através da biblioteca UIKit, fornecida pela Apple, é possível criar, configurar e manipular todos os componentes gráficos que precisamos. O *View Code* é muito mais flexível para customização e animação das *views*, alguns layouts mais complexos não seriam possíveis de serem implementados via *Interface Builder*. Por usar a mesma linguagem de desenvolvimento e ser totalmente procedural, o *View Code* torna possível o processo de *code review* e resolução de conflitos de *merge*, além de solucionar o problema da injeção de dependência no construtor e todo código ser analisado pelo compilador, acusando qualquer erro de tipo em tempo de compilação.

2.6 *Clean Code* e *Clean Architecture*

Clean Code e *Clean Architecture* são livros escritos por Robert Cecil Martin, que abordam conceitos e técnicas para desenvolvimento de software com qualidade. Mais do que isso, eles também advogam sobre a relação entre processo de desenvolvimento e qualidade final do produto.

O *Clean Code* aborda técnicas que facilitam a leitura e escrita de código, tornando a manutenção e adição de funcionalidades cada vez mais simples. O principal foco é ensinar a reconhecer e escrever boas funções, tendo como fundamento que um sistema é composto por um conjunto de funções e uma estrutura sólida é fundamental para apoiar toda qualidade do sistema.

O *Clean Architecture* de certa forma olha para um nível mais alto, agora não só preocupado com suas funções mas também em como elas são agrupadas e interagem entre si através de uma arquitetura. O autor apresenta os princípios de uma boa arquitetura, e propõe uma conhecida como *Clean Architecture*.

Esses conceitos foram essenciais no desenvolvimento do trabalho e influenciaram várias decisões de abordagens e implementações ao longo do desenvolvimento.

2.7 Scrum

Scrum é um dos principais *frameworks* utilizado para organizar e gerenciar projetos utilizando os valores e princípios do manifesto ágil. Pode ser combinado com outras ferramentas e métodos como um quadro *kanban* (SUTHERLAND, 2019), para representação dos estados e tarefas. Primeiro iremos apresentar algumas palavras chaves para o entendimento desse modelo e depois iremos introduzir um ciclo básico do seu funcionamento.

- ***Product Owner***: é a pessoa que define os itens que compõem o *Product Backlog* e os prioriza nas *Sprint Planning Meetings*. As tarefas devem ser divididas de forma que, no máximo, ocupem o tempo de uma *Sprint*;
- ***Scrum Team***: é a equipe de desenvolvimento;
- ***Product Backlog***: é uma lista contendo todas as funcionalidades desejadas para o produto;
- ***Sprint Backlog***: é uma lista de tarefas que o *Scrum Team* se compromete a fazer em um *Sprint*;
- ***Sprint Planning Meeting***: é uma reunião na qual estão presentes o *Product Owner* e todo o *Scrum Team*, nela o *Product Owner* descreve as funcionalidades de maior prioridade para a equipe e em conjunto definem o tempo necessário para cada atividade;
- ***Sprints***: representa uma janela de tempo dentro da qual um conjunto de atividades, elencadas no *Sprint Backlog*, devem ser executadas;
- ***Sprint Retrospective***: ocorre ao final de um *Sprint* e serve para identificar o que funcionou bem, o que pode ser melhorado e que ações serão tomadas para melhorar;
- ***Story Point***: é uma unidade subjetiva de estimativa utilizada por times ágeis para estimar o tempo de uma tarefa, a sua distribuição costuma levar em consideração que quanto mais pontos uma tarefa possui mais chance da estimativa estar errada, por isso é muito comum usar valores da sequência Fibonacci: 1, 2, 3, 5, 8.

O *Product Owner* faz a separação e priorização das tarefas atribuindo, *Story Points* que caiba na janela de tempo de uma *Sprint*, depois essas tarefas são cadastradas no *Product Backlog*, esse processo na maior parte das vezes ocorre no início do projeto.

A cada nova *Sprint* o *Product Owner* separa as tarefas que serão movidas para a *Sprint Backlog*. Em seguida, as tarefas são atribuídas na *Sprint Planning Meeting* para o *Scrum Team*. Ao final da divisão e discussão das tarefas, que podem ter seu *Story Point* alterado por observações do time, acontece a *Sprint Retrospective*, esse processo se repete

de acordo com o tempo definido em cada *Sprint*. Também é muito comum ocorrerem reuniões diárias rápidas para atualizar o time sobre o andamento das tarefas e resolver possíveis dúvidas ou impedimentos, essas cerimônias são chamadas de *Daily Meeting*.

2.8 Ferramentas

Após definir os conceitos, tivemos que encontrar ferramentas que nos auxiliassem a administrar e manter nossos *workflows*. Além de atender as demandas necessárias de cada fluxo é ideal que as ferramentas possam ser integráveis entre si e automatizadas. Em uma análise de custo vs. benefício, definimos as seguintes ferramentas para nosso projeto:

2.8.1 *Git Project*

Para os fluxos gerência e desenvolvimento usamos o *Git Project*,[19] um serviço Web que oferece diversas funcionalidades aplicadas ao git e a gestão de projetos. Para o fluxo de desenvolvimento cada funcionalidade foi definida como uma *milestone* e as tarefas dessa funcionalidade são *issues*. Ao criar uma *issue*, o *Product Owner* escreve uma descrição e atribui *Story Points*, com isso compomos nosso *Product Backlog*.

Para o fluxo de desenvolvimento criamos quadros representando os status da tarefa. Após a *Sprint Planning Meeting*, atribuímos uma *issue* a um desenvolvedor e ela entra no git project, para ser realizada durante a *Sprint*. Conforme a tarefa evolui ela tramita pelos quadros.

2.8.2 Git

Git é um sistema de controle de versão *open source* originalmente desenvolvido em 2005 por Linus Torvalds. Tendo uma arquitetura distribuída, o Git é um exemplo de um DVCS (*Distributed Version Control System*). Em vez de ter apenas um único local para o histórico de versões, como é comum em sistemas como CVS[20] e SVN,[21] no Git, a cópia de trabalho de cada desenvolvedor também é um repositório que pode conter o histórico completo de todas as alterações. Ele facilita o trabalho colaborativo entre desenvolvedores, para o nosso projeto utilizamos o *Gitflow* (fluxo Git) ilustrado na Figura 11.

Nessa configuração o *branch* (ramo) *master* representar o histórico de publicações do aplicativo, versões lançados na *AppStore* ou no *TestFlight*, o *branch develop* é uma ramificação da *master* e serve para integração das funcionalidades.

Para cada funcionalidade é criado uma *branch* a partir da *develop*, e todas as suas

[19] <https://github.com/features/project-management/>
[20] <https://pt.wikipedia.org/wiki/CVS>
[21] <https://pt.wikiversity.org/wiki/Subversion_-_SVN>

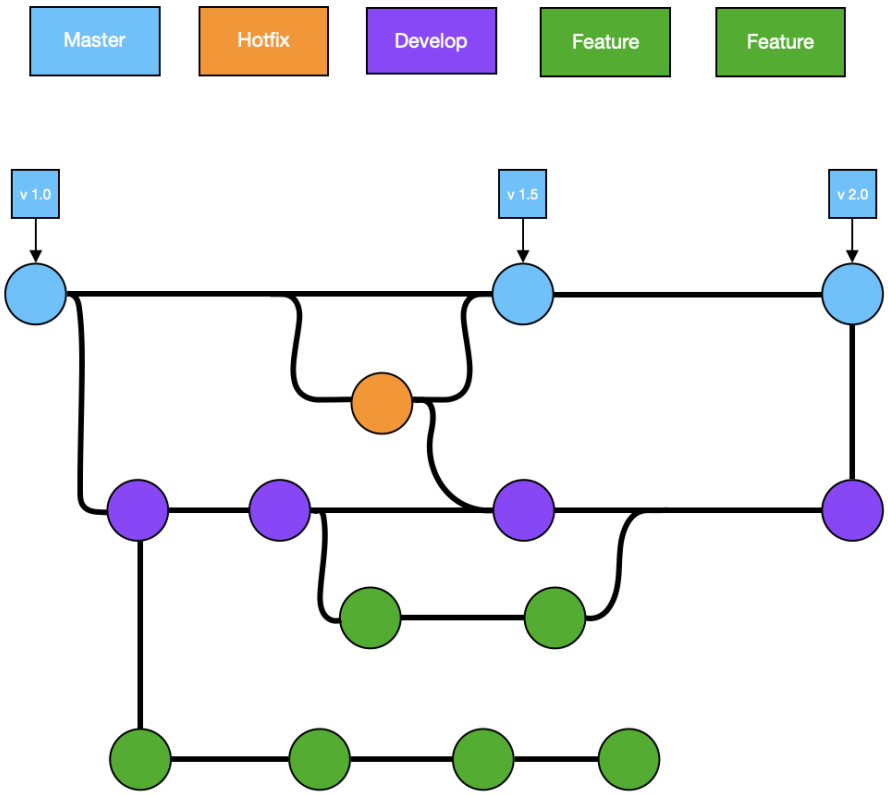

Figura 11 – Git Workflow.

tarefas tem como origem e fim a sua *branch* de funcionalidade correspondente, quando são desenvolvidas todas as tarefas de uma funcionalidade ela sofre *merge* (fusão) na *develop*. Quando temos um conjunto de funcionalidades que devem ir para produção, fazemos o *merge* da *develop* com a *master*, criando uma nova versão. Tarefas menores que não se encaixam no escopo de funcionalidade são desenvolvidas num *branch feature* fixo, seguindo o mesmo fluxo. Só existe um fluxo que foge desse padrão, que são os *hotfix* (correção de falhas que demandam certa urgência): neste caso é criada uma *branch* a partir da *master* e, feita a correção, fazemos o *merge* na *master*.

Do ponto de vista do desenvolvedor ele cria uma *branch* a partir da funcionalidade da sua tarefa e atribui um nome, seguindo o padrão `[<AbreviçãoDaFuncionalidade>-<Id-DaTarefa>]-<NomeDaTarefa>` e, após desenvolvida, realiza um *Pull Request* para *branch* de origem. Um *Pull Request* é uma maneira de informar à equipe sobre as alterações enviadas para uma *branch*. Pelo Git podemos automatizar algumas tarefas, como fazer a tramitação automática no git project entre algumas etapas, criar um *template* para cada vez em que é aberto um novo *Pull Request*, só permitir o *merge* após o código ser revisado por um número mínimo de pessoas, etc.

2.8.3 IDE (*Integrated Development Environment*)

Como IDE utilizamos o Xcode[22] que, atualmente, é o mais completo e avançado ambiente de desenvolvimento para produtos Apple. O Xcode foi lançado pela Apple em 2003 e contém um conjunto de ferramentas que auxiliam o desenvolvimento de software, para macOS, iPadOS, watchOS e tvOS. Nele podemos encontrar um conjunto de simuladores que permitem exibir os projetos, nos mais diversos aparelhos da fabricante.

Variando até mesmo a versão do sistema operacional, é possível testar várias situações e comportamentos, como mudar a localização, conexão à Internet ruim, pouca memória no dispositivo, rotação do dispositivo, etc. Outro destaque são os testes, sendo possível desenvolver e executar testes unitários e de interface. Existem diversos outras funções que nos permitem desenvolver com mais velocidade e praticidade, como *autocomplete*, documentação, visualização da hierarquia de *views*, construção de *layout* via *Storyboard*, entre outras.

2.8.4 Teste

Usamos o próprio Xcode como ferramenta para desenvolver nossos testes. Além de ser bem completo, uma ferramenta nativa evita problemas de suporte e dependências com bibliotecas de terceiros. Uma de nossas bases, para qualidade dos testes, será uma métrica fornecida pela própria IDE que verifica a porcentagem de código coberto por testes.

2.8.5 CI e CD

O *Bitrise*[23] em conjunto com scripts do *fastlane*[24] compuseram nossa ferramenta para implantar os serviços de CI e CD. A grande vantagem é a integração com o GitHub. A cada *Pull Request* aberto o *Bitrise* executa os testes automatizados e só aprova se todos forem executados com sucesso, caso contrário o *merge* é bloqueado. Ao fazer o *merge* da *develop* com a *master* e criar uma *tag* o CD pública o *build* na *AppStore*.

[22] <https://developer.apple.com/xcode/>.
[23] <https://www.bitrise.io>
[24] <https://github.com/fastlane/fastlane>

3 Apoio ao Desenvolvimento

Neste capítulo, apresentamos os conceitos e implementações de ferramentas que nos ajudaram a economizar tempo e manter a qualidade no processo de desenvolvimento. A Seção 3.1 discute sobre a importância de se adotar um padrão de código e exibe a ferramenta que adotamos para assegurar isso. Na Seção 3.2 mostramos o *template* desenvolvido para evitar o processo burocrático da criação de vários arquivos para compor uma cena. A Seção 3.3 relata como utilizamos e automatizamos o *Git Project* para gerenciamento das tarefas. A Seção 3.4 apresenta a nossa adaptação do Scrum utilizando o *Kanban* para auxiliar na gerência das atividades. Na Seção 3.5 apresentamos como utilizamos os *Targets* para resolver o problema de adicionar e gerenciar vários clientes em uma mesma base de código. Na Seção 3.6 discutimos a ferramenta de CI e CD e como ela foi utilizada no processo de desenvolvimento e publicação. Por fim, na Seção 3.7 apresentamos um resumo de todas ferramentas utilizadas no projeto assim como uma breve descrição do seu uso.

3.1 Padrão de Código

Um padrão de código consiste em definir regras e estilos para escrita de código, que devem ser adotados por todo o time a fim de manter a consistência em todas partes do projeto como, por exemplo, declarar primeiro os atributos depois as funções, inserir um espaço entre o nome da variável e seu tipo, etc. Padrões de código são muito importantes, pois programar é uma maneira de se comunicar, uma forma de se expressar, deixar um registro que muitas vezes será dividido com outras pessoas, com o compromisso de entender e evoluir a partir do último ponto. Definir padrões favorece uma comunicação rápida e eficaz, a partir desse momento a contribuição e entendimento se tornam mais simples.

Para implementar e assegurar os padrões utilizamos como ferramenta o *SwiftLint*.[1] Na sua configuração, ele possui um arquivo no qual é possível definir os conjuntos de regras e em quais *paths* elas se aplicam, a verificação é feita a partir da execução de um script que aplica todas regras aos arquivos adicionados e gera como saída *warnings* ou erros conforme o configurado. Para automatizar a verificação adicionamos a execução do script como um dos passos do *build* do Xcode de modo a gerar *warnings* ou erros para cada violação das regras de forma visual para o programador.

Outro ponto onde utilizamos o *SwiftLint* foi como um dos processos de verificação obrigatórios do GitHub[2] para autorizar o *merge* do *Pull Request*. Nessa etapa o script é executado pelo CI e, se algum erro for encontrado, ele é apontado como um comentário

[1] <https://github.com/realm/SwiftLint>
[2] <https://github.com>

e o *merge* da *branch* é bloqueado até que as pendências sejam resolvidas. Dessa forma conseguimos garantir a aplicação e consistência dos padrões em todo código.

3.2 *Template*

Quando definimos a arquitetura, nosso principal objetivo era garantir uma clara divisão de responsabilidade e desacoplamento entre as camadas e, para isso, utilizamos fortemente o conceito de inversão e injeção de dependências. Apesar de funcionar muito bem, há um preço a se pagar, um aumento na "burocracia" do código: por exemplo, para criar uma simples tela, são necessários no mínimo 7 arquivos, sendo o conjunto de todos arquivos necessários para se criar essa tela chamado de cena. Era extremamente maçante criar todos os arquivos e compor a injeção de dependência manualmente entre eles.

Nossa solução foi automatizar essa tarefa desenvolvendo um *template* para o *Xcode* chamado VIP, que funciona de forma similar a um *plugin* que pode ser baixado e instalado na IDE de qualquer membro da equipe. Após instalado, ele passa a ser listado em conjunto com as outras opções nativas já existentes. A Figura 12 ilustra seu uso: o desenvolvedor seleciona o *template*, insere o nome base de todos os arquivos e escolhe o *path*. Ao final do processo todos os sete arquivos contendo o código base da arquitetura de uma cena estarão disponíveis.

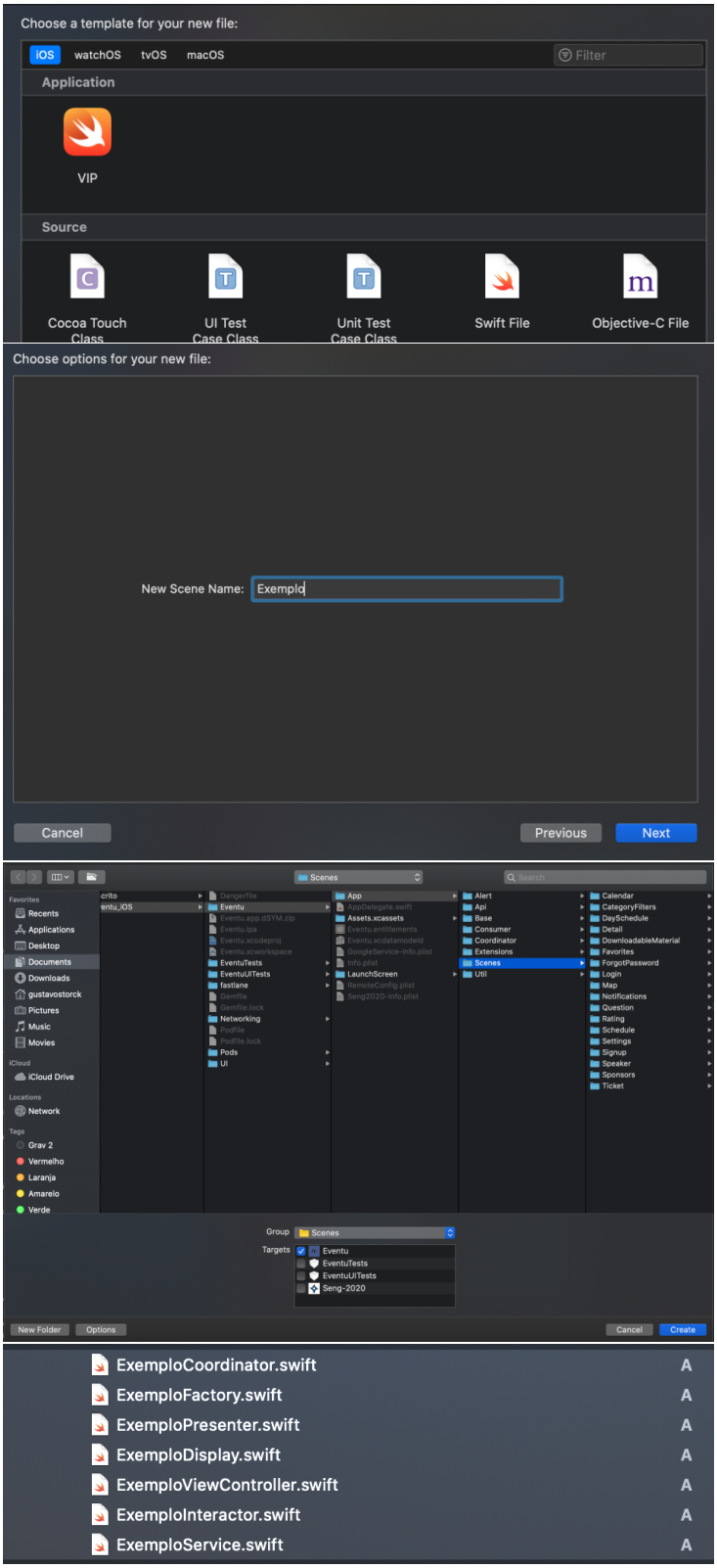

Figura 12 – Passos para criar os arquivos

3.3 *Git Project*

O *Git Project* foi escolhido como ferramenta de gerência para o desenvolvimento do projeto.

Todas as tarefas necessárias para o desenvolvimento do sistema foram cadastradas como *issues*, mencionadas anteriormente na Seção 2.8.1. *Issues* são entidades com título e descrição e que possuem uma seção que possibilita perguntas, discussões e comentários, além de ser possível atribuir alguns atributos como rótulo e responsável pela tarefa. Essas *issues* são transformadas em cartões para facilitar a visualização dentro do quadro do *Git Project*.

O Quadro contém todas os estados do desenvolvimento e foi automatizado para quando uma *issue* fosse aberta, ela fosse adicionada à coluna de *Product Backlog*. No início de cada semana as tarefas com maior prioridade e que podem ser desenvolvidas dentro do *time box* da *sprint* são movidas para o *To do*. A coluna de *Done* também foi automatizada para mover um cartão quando o código relativo a uma determinada *issue* for incorporado ao ramo de origem. Os demais estados exigem que progressão das tarefas seja feita manualmente pelo seu responsável.

3.4 Fluxo de Gestão

O *Scrum* traz conceitos de gerência eficazes e flexíveis, podendo ser adaptados a vários contextos e tamanhos de empresas. Para o nosso projeto, foi levado em consideração que todos os papéis serão desempenhados por uma única pessoa, por isso não foram exercidos todos os passos do modelo tradicional e o fluxo foi adaptado para nossa realidade.

Nosso fluxo começa após a concepção do sistema, definindo o objetivo geral e funcionalidades pretendidas. Nesse ponto como *Product Owner* fiz uma separação das tarefas macro: primeiro são divididas as funcionalidades e tarefas gerais do sistema, como login, camada de serviço, tela de palestrantes, etc. É interessante destacar que nem todas as tarefas serão definidas no início do projeto, podendo surgir novas tarefas ao decorrer do desenvolvimento, como uma alteração de escopo exigida pelo cliente (novas funcionalidades), ou tarefas para correção de erros.

Em seguida as tarefas macro são divididas em tarefas micro, que contém uma descrição mais técnica e uma estimativa em *Story Points*. Cada *Story Point* equivale a 1 hora e meia de trabalho e nossa *Sprint* tem duração de 1 semana. Após a identificação e classificação das tarefas elas são cadastradas no *Product Backlog*. O *Product Backlog* faz parte do nosso fluxo de desenvolvimento.

O fluxo de desenvolvimento consiste nas sequência de etapas a qual uma tarefa deve ou pode passar para se tornar efetivamente concluída, gerenciadas através do Git

Project como um *kanban*. Essas etapas são:

- ***Product Backlog***: Lista de todas tarefas necessárias para se cumprir o projeto
- ***To do***: tarefas que foram atribuídas a um desenvolvedor na *Sprint Planning Meeting*, mas ainda não começaram a ser realizadas;
- ***In progress***: tarefas em desenvolvimento;
- ***On Hold***: tarefas que tiveram algum impedimento, que impossibilitaram a continuação do desenvolvimento;
- ***Done***: uma tarefa é considerada concluída quando ela passa por pelo menos *In progress*, *Code Review* e *Test*.

Esse fluxo ajuda a assegurar a qualidade e transparência do desenvolvimento, além de dar visibilidade a cada tarefa.

3.5 *Targets*

Todo projeto contém pelo menos um *target*. Um *target* especifica uma aplicação, define um nome, recursos dos dispositivos móveis que irão executar o código e os arquivos que o compõem. O Xcode permite criar múltiplos *targets* pertencentes ao mesmo projeto. Isso é muito importante devido ao nosso modelo de negócios, pois nossa proposta é oferecer a mesma base de código para múltiplos clientes, fazendo personalizações simples.

Na primeira versão do projeto mantínhamos um repositório para cada cliente, faziamos um *fork* do código base e customizávamos para atender a demanda do evento. Ao longo do tempo esse tipo de abordagem ficou totalmente inviável, pois se descobríssemos um erro tínhamos que atualizar todos os repositórios, resultando em uma grande perda de tempo, pois o número de repositórios era enorme.

Analisando melhor, percebemos que as atualizações se limitavam ao nome, logomarca, URL do servidor e escolha de funcionalidades. A escolha de funcionalidade poderíamos resolver utilizando o *Remote Config*, porém tanto o *Remote Config* quanto a publicação na *AppStore* exigiam que cada aplicativo tivesse um identificador único, conhecido como *BundleID*, o que nos motivou inicialmente a usar projetos distintos para cada evento. No entanto, com o uso dos *targets* foi possível solucionar nosso problema: cada *target* tem seu próprio *BundleID* e pode sofrer *deploy* separadamente. Podemos definir constantes para cada *target* o que permite definir imagem, nome e URL específicas de cada evento. Hoje, para cada evento criamos um novo *target*, definimos *BundleID*, imagem, URL, e um novo projeto no Firebase, serviço responsável por gerenciar a autenticação, *Remote Config*, logs de eventos e falhas. Tudo se encontra dentro do mesmo projeto e

repositório, com todos arquivos fontes compartilhados por todos os eventos. A Figura 13 mostra uma lista de *targets* do projeto.

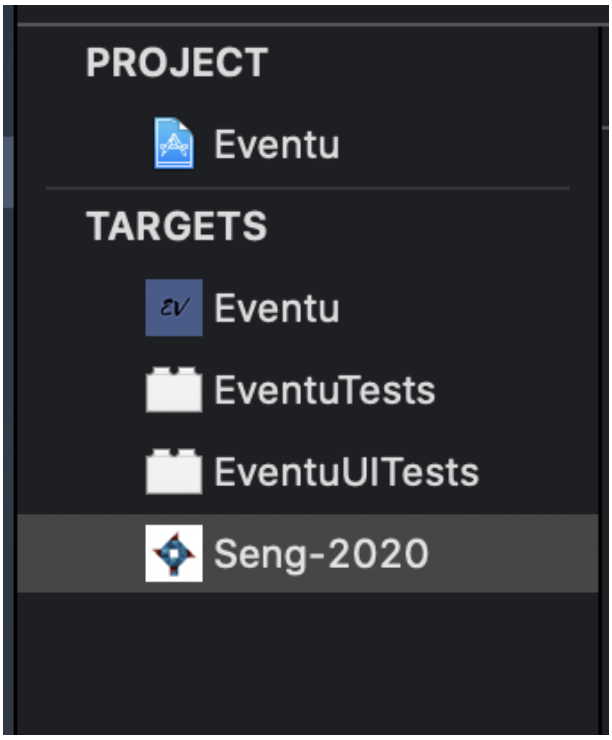

Figura 13 – Diferentes Targets

3.6 CI e CD

CI e CD são ferramentas fundamentais que compõem nossa esteira de desenvolvimento e *deploy*. No projeto foram configurados 3 *workflows*.

- ***developer***: é executado ao abrir um *pull request* de qualquer *branch* para *developer*, sendo responsável por fazer o *buid* do App, rodar os testes unitários e verificar o padrão de código. Depois de ter um *pull request* aberto, qualquer novo *push* executa o fluxo novamente;
- ***master***: tem as mesmas etapas da *developer*, mas é executada ao abrir um *pull request* da *developer* para *master*;
- ***deploy***: executa ao se criar uma nova Tag para o projeto, além de executar todas verificações feitas pela *developer*, envia e publica o aplicativo na *App Store*.

Os fluxos de *developer* e *master* fazem parte do CI, e estão integrados com o GitHub. Habilitam o botão de *merge* caso todos os passos sejam executados com sucesso, como mostra a Figura 14. O fluxo de *deploy* caracteriza o CD, automatizando todo o processo de publicação do App. Para compor esses fluxos foram utilizadas 3 ferramentas.

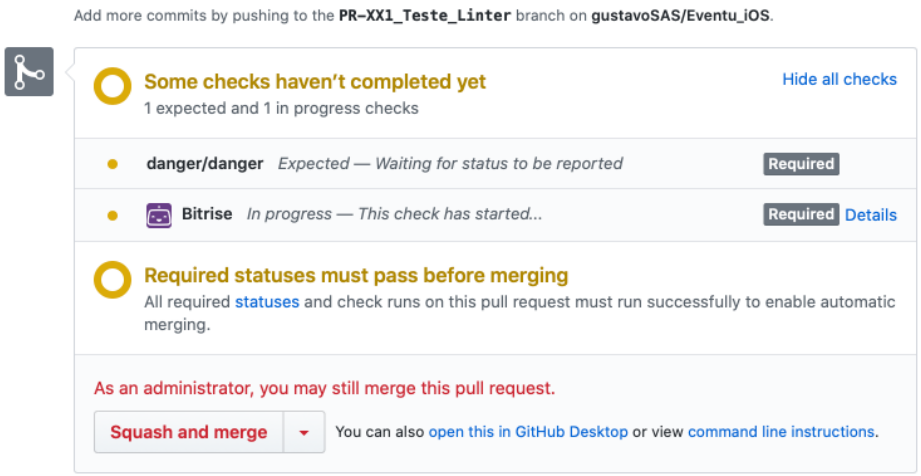

Figura 14 – Status do Github ao aguardar as verificações do CI.

- ***Fastlane***: é uma ferramenta gratuita, *open source*, com diversos *plugins* escritos pela comunidade. Foi utilizado para executar os testes e fazer o processo de publicação do aplicativo. Uma das características mais interessantes para o nosso projeto é que ele não depende de estar em um serviço externo para ser executado: depois de configurado, podemos rodar as *lanes* localmente. A Listagem 3.1 exibe o arquivo de configuração do *Fastlane*;

- ***Danger***: é uma ferramenta gratuita, *open source*, responsável por automatizar as convenções da sua equipe. No projeto, ele verifica o tamanho do *Pull Request*, limitado a 500 linhas de adição de código, e faz a análise do padrão de código utilizando o *Linter*. Caso algum problema seja encontrado ele comenta diretamente trecho de código correspondente pelo GitHub. A Figura 15 mostra um comentário feito pelo Danger após ser executado pelo CI e encontrar uma divergência no padrão de código;

Listagem 3.1 – Arquivo de configuração Fastlane

```
# Uncomment the line if you want fastlane to automatically update itself
# update_fastlane

default_platform(:ios)

platform :ios do
  desc "Create app on Apple Developer and App Store Connect sites"
  lane :create_app do
    produce
  end

  desc "Match Dev"
  lane :match_dev do
    match(
      type: "appstore",
      app_identifier: [
        "br.com.gsas.Eventu"
      ]
    )
  end

  desc "Run App for beta"
  lane :beta do
    sync_code_signing(type: "appstore")

    gym(
      workspace: "Eventu.xcworkspace",
      scheme: "Eventu",
      include_bitcode: true,
      export_options: {
        method: "app-store",
        provisioningProfiles: {
          "br.com.gsas.Eventu" => "match AppStore br.com.gsas.Eventu",
        }
      }
    )

    pilot(
      skip_waiting_for_build_processing: true,
      beta_app_review_info: {
        contact_email: "gustavo.storck1@gmail.com",
        contact_first_name: "Gustavo",
        contact_last_name: "Storck",
        contact_phone: "27997053363",
        demo_account_name: "teste",
        demo_account_password: "teste12345",
        notes: "Essa é uma versão do Eventu apontando para o ambiente de Homologação, para fins de teste."
      }
    )
  end

  desc "Run Tests"
  lane :tests do
    scan(
      workspace: "Eventu.xcworkspace",
      devices: ["iPhone 11 Pro Max"],
      scheme: "Eventu",
      reinstall_app: true,
      clean: true
    )
  end
end
```

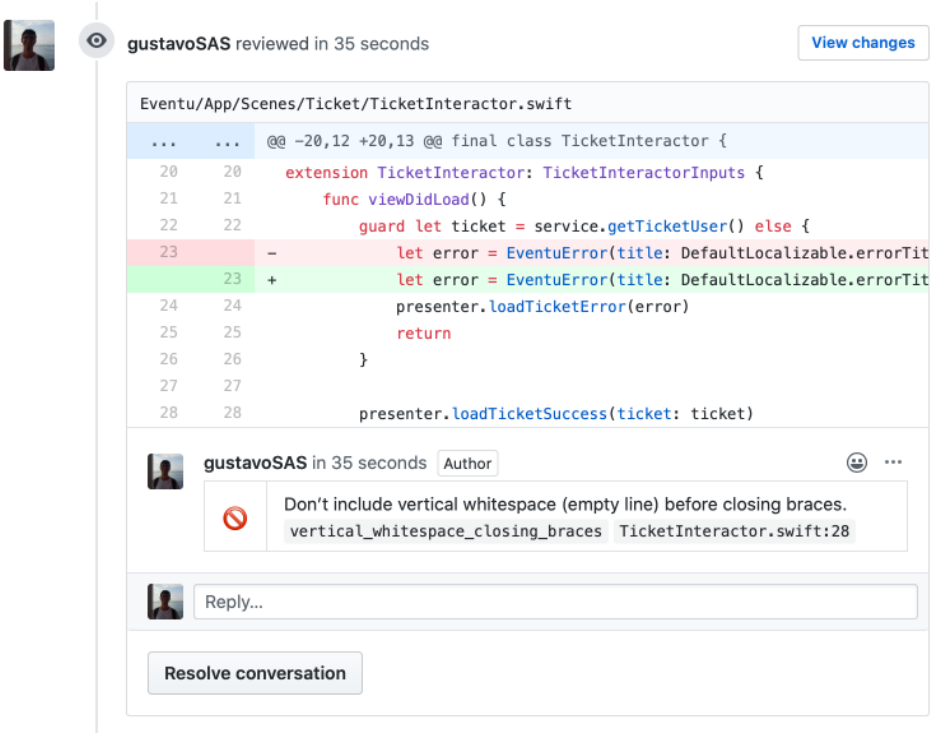

Figura 15 – Comentário *Danger*

- ***Bitrise***: é um Serviço externo de CI/CD pago, ele se conecta ao repositório, define os *triggers*, que determinam qual *workflow* será executado quando determinada ação ocorrer, além de habilitar ou não o processo de *merge*. Executa o *Fastlane* e o *Danger* para implementar as ações de teste, análise de código e publicação. A Figura 16, exibe o progresso da execução do CI, iniciado após um *pull request* para *developer*.

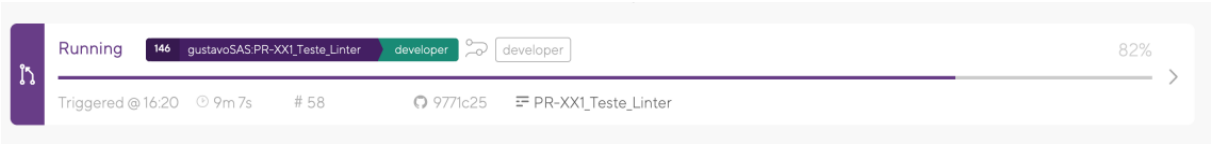

Figura 16 – *Status Bitrise*

3.7 Resumo das Ferramentas

A Tabela 1 lista todas as ferramentas descritas neste capítulo, resumindo em uma breve descrição para que foram utilizadas no contexto deste trabalho.

Tabela 1 – Tabela de Ferramentas usadas no projeto.

Nome	**Uso**
Astah <https://astah.net/>	Ferramenta utilizada para modelagem UML e de dependências.
Figma <https://www.figma.com/>	Ferramenta de *design* para prototipação das telas e dos fluxos.
GitHub Project <https://github.com/features/project-management/>	Ferramenta para gerência de projeto, usado para registrar as tarefas e como Kanban.
GitHub <https://github.com/>	Ferramenta para controle de versão do código.
Fork <https://git-fork.com/>	Cliente *desktop* para Git.
Xcode <https://developer.apple.com/xcode/ide/>	Ambiente de desenvolvimento para ecossistema Apple.
Proxyman <https://proxyman.io/>	*Proxy* de depuração Web, para mudar as respostas do servidor. Usada nos testes exploratórios.
CocoaPods <https://cocoapods.org/>	Gerenciador de dependências para projetos Swift e Objective-C.
SwiftLint <https://github.com/realm/SwiftLint>	Ferramenta para aplicar o estilo e as convenções do Swift.
Fastlane <https://fastlane.tools/>	Ferramenta para automatizar o processo de desenvolvimento e publicação
Danger <https://danger.systems/swift/>	Automatiza tarefas de revisão de código no CI.
Bitrise <https://www.bitrise.io/>	Ferramenta de CI/CD.

4 Especificação de Requisitos

Dentro do escopo que um projeto de graduação permite, o objetivo principal deste trabalho foi enfrentar o desafio de aprender e aplicar o desenvolvimento de software com qualidade, sendo qualidade, para nós, um objetivo alcançado através da garantia de três fatores: testabilidade, manutenibilidade e escalabilidade. Durante todo processo de desenvolvimento, tentei usar conceito de *clean code* (MARTIN, 2009), *clean architecture* (MARTIN, 2018), SOLID, escrita de testes e preparar uma *pipeline* de desenvolvimento com, Git, CI, CD, etc. Neste sentido, as atividades de especificação e análise de requisitos foram contempladas de maneira mais simplificada, permitindo que houvesse tempo para o desenvolvimento dentro do escopo do projeto.

Esse capítulo destina-se a mostrar a Especificação de Requisitos da Eventu, a Seção 4.1 apresenta a descrição do mini-mundo, envolvendo todo sistema da Eventu, site e aplicativo. A Seção 4.2 apresentamos as tabelas de requisitos referente ao aplicativo. Na Seção 4.3 temos o diagrama UML de todo o sistema, com explicações sobre as classes e relacionamentos. Por fim a Seção 4.4 trata sobre a prototipação das telas do aplicativo.

4.1 Descrição de Mini-mundo

Este trabalho está inserindo no contexto da plataforma Eventu, um sistema para organização e gerência de eventos. Por meio dela é possível criar e personalizar um site e aplicativo nativo (Android e iOS) para um evento. Existem 3 atores no sistema:

- Contratante: cliente da Eventu, é quem solicita o serviço, define o escopo e funcionalidades gerais. A visão dele é usada para personalizar a identidade visual do produto. Também pode solicitar que sejam desenvolvidas funcionalidades específicas;
- Gerente: administradores do evento, definidos pelo contratante. Possuem a função de editar, inserir informações, cadastrar preço, número de vagas, ativar funcionalidades, visualizar estatísticas e questionamento dos usuários, enviar notificações *push* para o *App*, enviar e-mails. Todas as atividades do gerente são realizadas pelo site. Podem ter papéis limitados, permitindo apenas um conjunto de ações no sistema;
- Usuário: participantes do evento, que se cadastram no evento, nas atividades, realizam o pagamento e consomem o conteúdo oferecido.

A interação desses atores é feita por meio do:

- Site:
  - Para o gerente: possibilita toda interação com o sistema, através de um *dashboard* é possível personalizar e obter estatísticas do evento, por ele são feitas as configurações que devem refletir tanto no site quanto nos aplicativos;
  - Para os usuários: é possível criar uma conta, escolher as atividades que pretende participar, comprar o ingresso e visualizar informações básicas do seu perfil, como atividades cadastradas e status da transação.
- Aplicativos: são destinados aos participantes do evento, nele deve ser possível visualizar todas as atividades do evento, podem ter meios de avaliação, localização, perguntas e materiais para *download*, permitir que o usuário favorite, adicione ao seu calendário. Os *Apps* permitem que o usuário faça login, que é integrado ao cadastro feito pelo site, sendo possível ver, a partir desse momento, informações específicas como número do seu ingresso, eventos escolhidos, etc.

Alguns pontos a serem considerados: o sistema deve ser personalizável para se adequar às características do evento, possuindo o máximo de automatização. Existem algumas configurações adicionais como inserir pontos de interesse da cidade e opção de telas para patrocinadores. É um requisito fundamental que o app respeite o *Design System*, conjunto de recomendações para criar uma interface, de cada plataforma.

4.2 Tabela de Requisitos

Nesta seção encontram-se as tabelas de requisitos funcionais (Tabela 2), regras de negócio (Tabela 3) e requisitos não-funcionais (Tabela 4), que serviram de base para planejamento do *Product Backlog*.

Todo o sistema (site, iOS e Android) foi implementado, mas dentro do escopo deste trabalho mostramos apenas os requisitos e implementações referentes ao aplicativo iOS.

Tabela 2 – Tabela de Requisitos funcionais do aplicativo iOS.

Id	**Descrição**	**Prioridade**	**Relacionado**
RF01	O aplicativo deve mostrar em sua inicialização a logo e o nome do evento.	Alta	
RF02	O aplicativo deve mostrar a lista de atividades do evento, separadas por dia e ordenadas por hora de início.	Alta	
RF03	O aplicativo deve permitir que as atividades sejam filtradas por uma ou mais categoria.	Média	RF02
RF03	O aplicativo deve exibir a lista com todos os palestrantes do evento, incluindo nome, foto e descrição.	Alto	
RF04	O aplicativo deve exibir e armazenar notícias enviadas pelos organizadores do evento.	Alto	
RF05	O aplicativo deve permitir o cadastro de usuários, desde que o administrador tenha habilitado esta opção.	Alta	
RF06	O aplicativo deve exibir locais de interesse escolhidos pela organização.	Alta	
RF07	O aplicativo deve permitir que os locais sejam filtrados por categoria.	Média	RF06
RF08	O aplicativo deve permitir que o usuário favorite atividades.	Média	RF02
RF09	O aplicativo deve permitir que o usuário inclua atividades em seu calendário.	Média	RF02
RF10	O aplicativo deve exibir atividades nas quais o usuário se cadastrou pelo site no seu calendário.	Alta	
RF11	O aplicativo deve permitir que o usuário faça login, por e-mail ou algum outro provedor de serviço, e mostre algumas informações básicas, como atividades escolhidas e número do ingresso.	Alta	RF10, RF02
RF12	O aplicativo deve permitir que o usuário avalie atividades.	Média	RF11, RF02
RF13	O aplicativo deve permitir que o usuário interaja com uma atividade enviando perguntas.	Média	RF11, RF02
RF14	O aplicativo deve exibir informações de uma atividade, como descrição, palestrante, local, horário, materiais para download.	Alta	RF02
RF15	O aplicativo deve permitir que o usuário selecione um alarme para avisar quando a atividade está preste a ocorrer.	Média	RF14, RF02
RF16	O aplicativo deve exibir os patrocinadores de um evento, separados por cota ou categorias.	Alta	
RF17	O aplicativo deve gerar estatísticas sobre o evento.	Alta	
RF18	O aplicativo deve permitir a recuperação de senha da conta do usuário.	Alta	

Tabela 3 – Tabela de Regras de Negócio referentes ao aplicativo iOS.

Id	**Descrição**	**Prioridade**	**Relacionado**
RN01	As avaliações dadas a um atividade podem variar de 1 a 5 estrelas.	Alta	RN02
RN02	O usuário só precisa estar autenticado quando for avaliar uma atividade, fazer uma pergunta ou olhar informações do seu perfil, como foto e número de ingresso, se houver.	Alta	
RN03	A pergunta feita por um usuário na atividade pode ter até 200 caracteres.	Alta	RN02
RN04	Os valores do alarme são pré definidos em: 15, 30 ou 45 minutos antes do início da atividade.	Alta	
RN05	O número do ingresso do usuário deve ser exibido em texto plano e QRcode.	Alta	RN02
RN07	As atividades devem ser exibidas em ordem crescente de horário inicial.	Alta	

Tabela 4 – Tabela de requisitos não funcionais para o aplicativo iOS.

Id	**Descrição**	**Prioridade**	**Relacionado**
RNF01	O aplicativo deve permitir uma alta personalização, com a maior parte das informações sendo providas pelo *back-end*.	Alta	
RNF02	A interface do aplicativo deve ser simples e intuitiva, seguindo as *guidelines* de cada sistema.	Alta	
RNF03	Os aplicativos devem manter grande parte das suas funções mesmo sem acesso à Internet.	Alta	
RNF04	Deve ser implementado um sistema de log que acompanhe *crashs* que ocorram na aplicação.	Alta	
RNF05	O aplicativo deve permitir a inserção e customização de um novo evento/cliente de maneira facilitada, em menos de 1 hora.	Alta	
RNF06	Deve ser possível disponibilizar ou não funcionalidades sem a necessidade de atualizar uma nova versão do aplicativo.	Alta	
RNF07	Deve ser possível enviar Push Notification para toda base de usuários.	Alta	
RNF08	O código fonte deve ser o mesmo para todos os clientes, deixando apenas como exceção alguns arquivos de configuração.	Alta	
RNF08	O sistema deve gerar log sobre ações e informações dos usuários.	Alta	

4.3 Diagrama do Sistema

Apesar do escopo deste trabalho ser o aplicativo iOS, a Figura 17 apresenta o diagrama de classes referente a todo o sistema da Eventu, de modo que se possa ter um entendimento geral dos conceitos do domínio no nível de requisitos. A seguir, explicamos cada uma das classes do diagrama.

Pessoa_Juridica, representa o contratante do sistema, serve apenas para a mantermos um registro sobre nossos clientes, e **Pessso_Fisica** que pode ser tanto um usuário quanto um administrador.

Um **Administrador** possui a função de gerenciar o evento ao qual foi atribuído: ele quem define as atividades, os conteúdos e visualiza os *logs*. Já o **Usuario** representa o ator principal do nosso sistema, que interage com o aplicativo e suas funções: o atributo *ticket* representa um id único atribuído no momento da inscrição, pode ser usado pelos administradores para liberar acesso, lista de presença, etc.

A classe **Noticia** representa uma informação, como um *feed* de notícias, pertencente a um evento criada pelo administrador que pode ser visualizada pelos usuários deste evento.

A classe **Evento**, é o produto que foi alugado pelo nosso cliente e é gerenciado pelo administrador, todo evento possui um nome e data para seu início e fim. Em cada evento também podem ser listadas as empresas que compraram as cotas de patrocínio do evento, esses patrocinadores são únicos de cada evento, o sistema não permite que sejam compartilhados entre múltiplos eventos ou clientes. Por exemplo a cota ouro foi comprada pelas empresa A, B e a cota bronze pelas empresas X, Y. As cotas são representadas pela classe **Cota_Patrocinio** e as empresas patrocinadores pela classe **Patrocinio**.

Inscricao representa quais atividades o usuário escolheu em determinado evento. Possui o atributo valor, que representa o total pago, somando as atividades escolhidas e o valor fixo do evento. Somente pode estar associada à **Atividade** que pertence ao **Evento** do qual ela faz parte.

Pergunta, **Alarme**, **Avaliacao** são classes que representam a interação direta do usuário com uma atividade. Uma pergunta é uma duvida sobre a atividade; o usuário pode definir um alarme para cada atividade, para ser avisado antes do seu inicio; e avaliação é uma nota de 1 a 5 atribuída a uma atividade, sendo que o usuário pode avaliar a atividade mais de uma vez, mas apenas a última nota será válida.

Atividade é a classe responsável por descrever o que terá em um evento, como uma palestra, alimentação, visita técnica, etc. O que será visível em cada atividade é representado pelos parâmetros que começam com **show**. Contém um conjunto de informações úteis ao usuário, dentre elas o **Aviso** que descreve duração, pré-requisitos e linguagem da atividade. Também é possível definir o **Local** que a atividade ocorrerá, sendo que os locais possuem

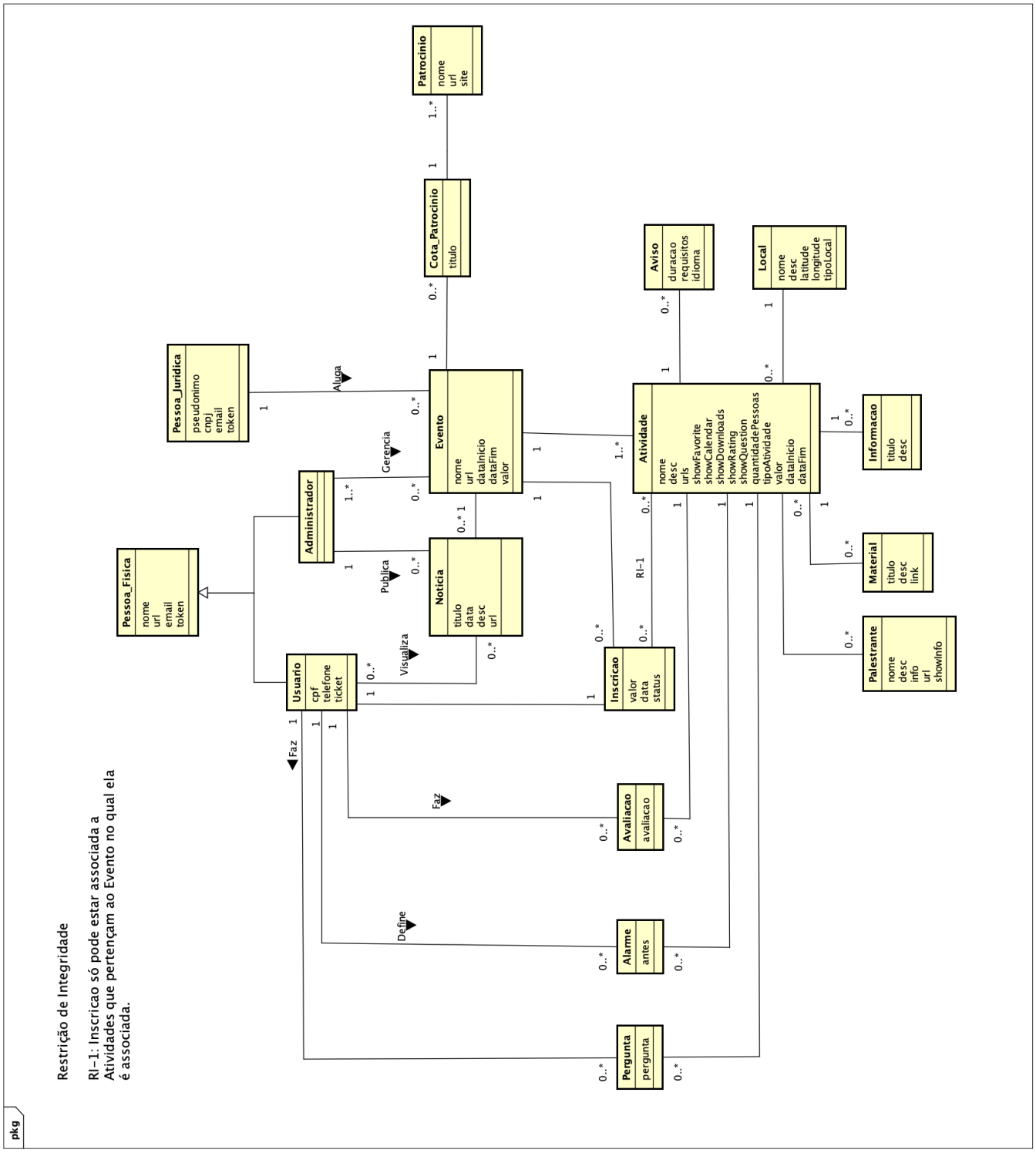

Figura 17 – Diagrama de classes UML relativo ao sistema Eventu completo (site e aplicativos).

tipos para podermos fazer pesquisas com filtros. Uma atividade pode conter uma lista de palestrantes representada pela classe **Palestrante**; uma lista de **Material** que podem ser obtidos pelos usuário; e uma lista de informações genéricas com título e descrição, contendo mais detalhes sobre o evento, representada pela classe **Informacao**.

4.4 Prototipação das Telas

A prototipação é uma forma visual, de verificar se nossa interface está de acordo com os requisitos definidos no sistema, além de poupar tempo de desenvolvimento, pois temos de antemão a relação de todos os componentes visuais e suas especificações. A prototipação foi dividida em 3 etapas e utilizou-se da ferramenta Figma.[1] Na primeira, representada na Figura 18, definimos os componentes, cores e tipografia, de modo que seja possível criar telas mais coesas e reaproveitáveis.

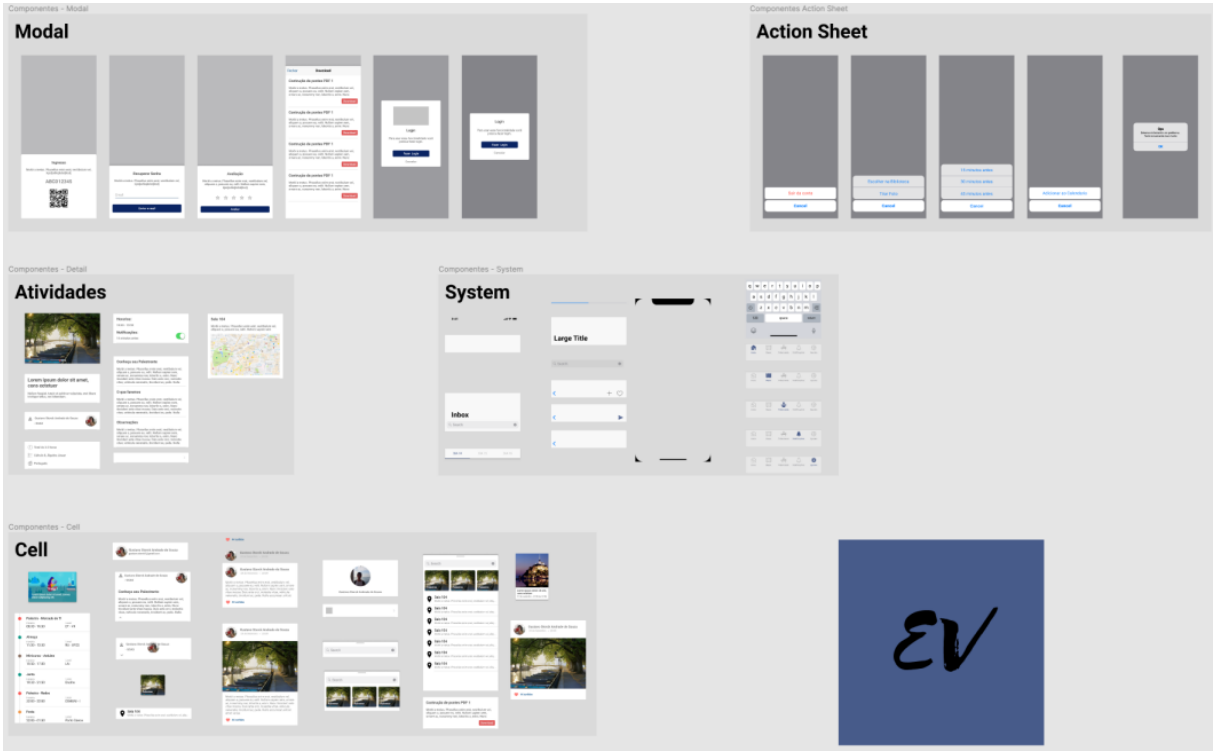

Figura 18 – Prototipação Componentes

O segundo passo foi projetar todas as telas de forma a abranger as funcionalidades e comportamentos especificados. Um exemplo de resultado encontra-se na Figura 19, representando a prototipação das telas usando os componentes anteriormente apresentados. Essas telas visam atender os requisitos estabelecidos e auxiliar na implementação do *layout*.

[1] <https://www.figma.com>

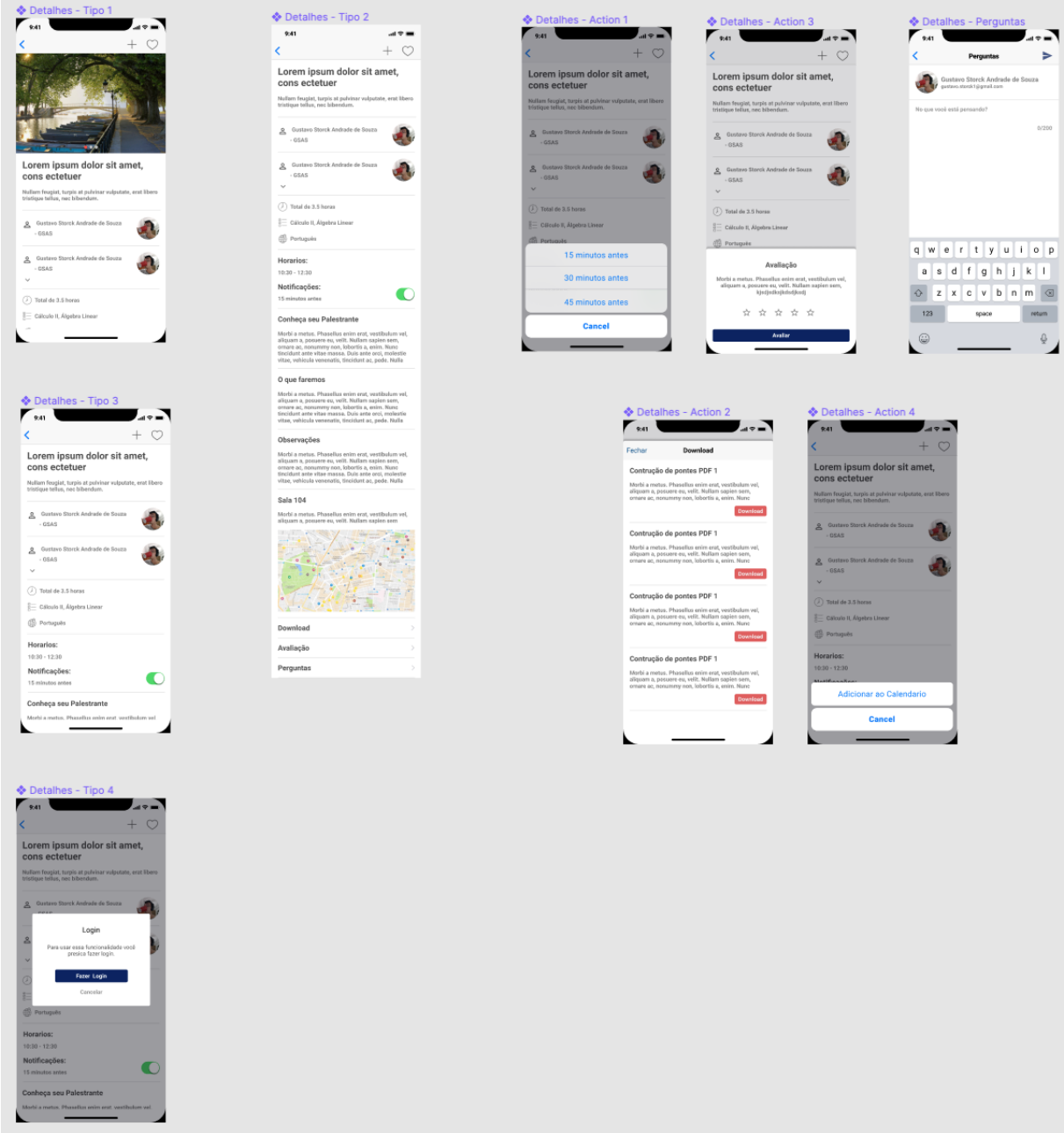

Figura 19 – Exemplo Prototipação Telas

Por último, estruturamos os fluxos do App contendo variações de *layout* que poderiam ocorrer em cada contexto. Como exemplo, o fluxo de detalhes de um evento é exibido na Figura 20, com a prototipação dos fluxos conseguimos ver as variações das ações e interfaces de acordo com a regra de negócio e opções escolhidas pelo usuário.

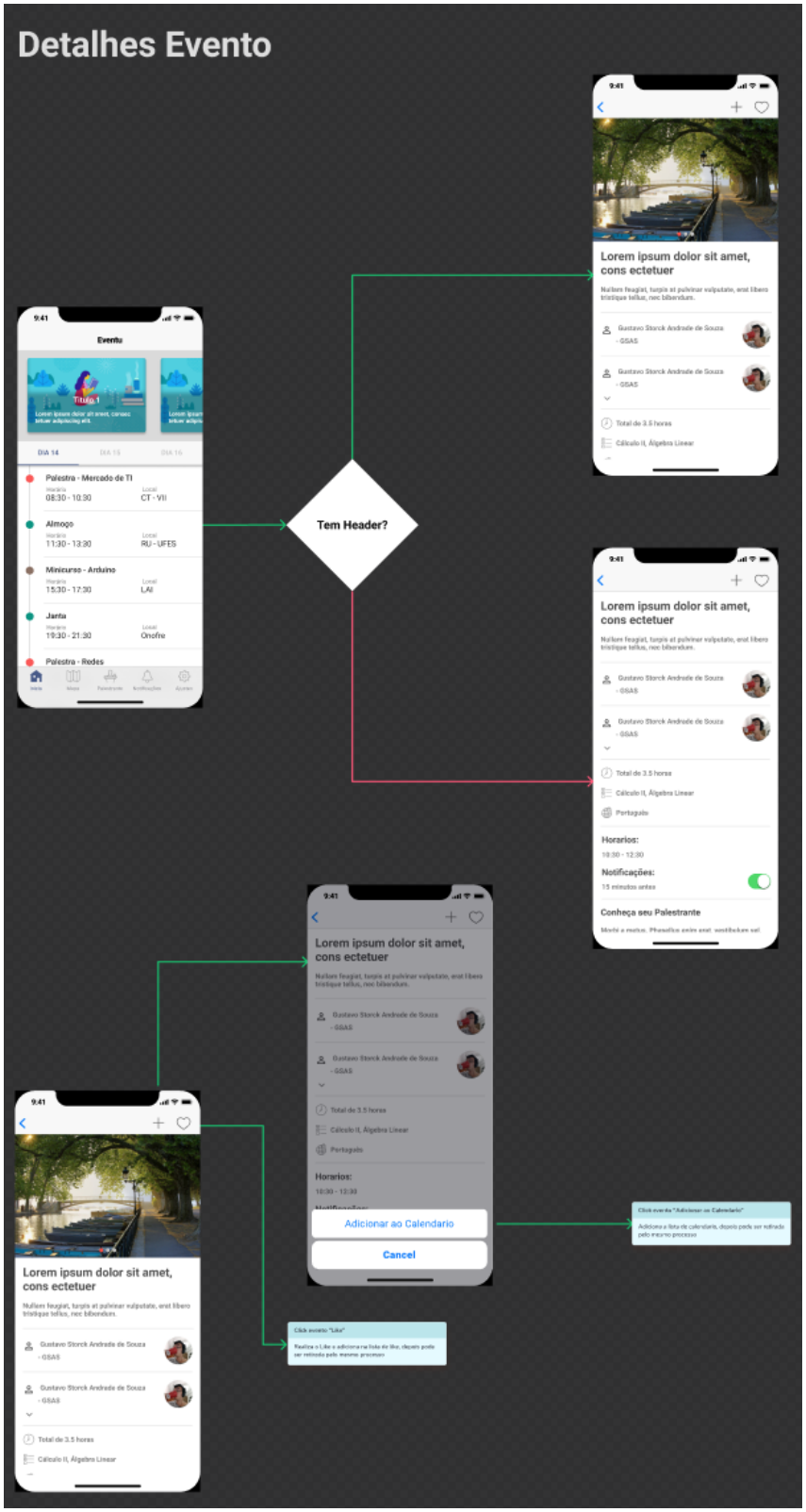

Figura 20 – Exemplo Prototipação Fluxos

Os artefatos produzidos na prototipação ajudaram a compor a descrição e tarefas do *Product Backlog*. Com o conhecimento de todas as telas e fluxo, conseguimos criar tarefas referentes à composição gráfica das interfaces e pudemos usar como recurso para

auxiliar o programador no desenvolvimento das *views* e fluxos.

5 Projeto e Implementação

Neste capítulo discutimos sobre as escolhas de implementação feitas durante o trabalho. A Seção 5.1, exibe os módulos e arquitetura utilizadas no projeto. A Seção 5.2 mostra o processo que utilizamos na implementação das *views*. Na Seção 5.3 apresentamos os tipos de testes feitos e quais padrões foram adotados. A Seção 5.4 apresenta as bibliotecas do Firebase que foram adotadas com a respectiva discussão sobre sua utilização. E por fim na Seção 5.5 é apresentado todo o sistema por meio de uma série de capturas de telas.

5.1 Módulos

Módulos são definidos por uma unidade separada de projeto, eles podem ser desenvolvidos e distribuídos de forma independente e serem reutilizados por diversas aplicações. A Figura 21 exibe todas dependências externas que temos no *App* da Eventu. Os módulos **UI** e **Networking** foram criados para atender ao domínio específico da Eventu, por isso são de caráter privado se integram ao projeto como qualquer outro módulo de terceiros. O módulo de **UI** reúne os componentes gráficos que foram criados especificamente para Eventu, fornecendo implementações comuns de *layout*, contendo todos os componentes usados no aplicativo, além de fornecer recursos como cor, tipografia e espaçamento, e encapsular algumas bibliotecas externas como a de carregamento de imagens e de animação. O segundo é o módulo de **Networking**, responsável por fornecer a implementação de acessos a recursos da Internet usando o protocolo *HTTP* e o estilo arquitetural *REST (PEREIRA, 2016)* utlizando o Alomofire.[1]

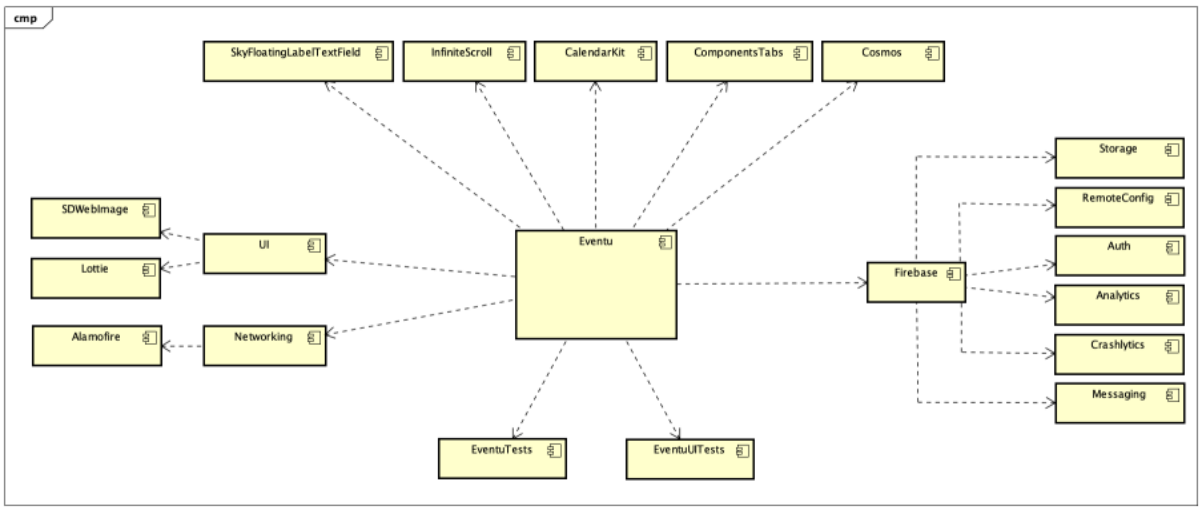

Figura 21 – Dependências externas

[1] <https://github.com/Alamofire/Alamofire>

EventuTeste e **EventuUITests**, são os módulos responsáveis por executar os testes unitários e de interface, respectivamente. Os módulos relativos ao **Firebase** foram de grande importância para atendermos as especificações do projeto e serão explicados em mais detalhes na Seção 5.4. Por fim, os módulos localizados no topo da imagem, são implementações visuais de algum componente que despenderiam muito tempo para serem feitos por conta própria. Todos os módulos externos antes de serem adotadas passaram por uma avaliação de documentação, suporte e apoio da comunidade. Tentamos isolá-los da melhor forma possível para que, caso exista a necessidade de substituição, isso seja feito com a maior segurança e no menor tempo possível.

Adicionalmente também foi desenvolvido um terceiro módulo, chamado *ProgressStepView*,[2] que fornece a implementação de um componente gráfico, não disponível nativamente no iOS, de controle de progresso animado. Ele não é usado no aplicativo, a sua concepção e publicação foi para demonstrar os conceitos aprendidos durante esse trabalho na criação, utilização e distribuição de módulos usando o gerenciador de dependência *CocoaPods*.[3] Esse módulo está disponível da mesma forma que outras 73 mil bibliotecas criadas pela comunidade de desenvolvimento iOS e por grander empresas como Google[4] e Airbnb.[5]

A Figura 22 exibe a divisão interna das funcionalidades, cada funcionalidade é composta por uma ou mais telas, cada tela é desenvolvida baseada em uma arquitetura, que chamamos de *cena*, que provê a implementação e divisão das responsabilidades.

[2] <https://github.com/gustavoSAS/ProgressStepView>
[3] <https://cocoapods.org/>
[4] <https://github.com/airbnb/lottie-ios>
[5] <https://github.com/airbnb>

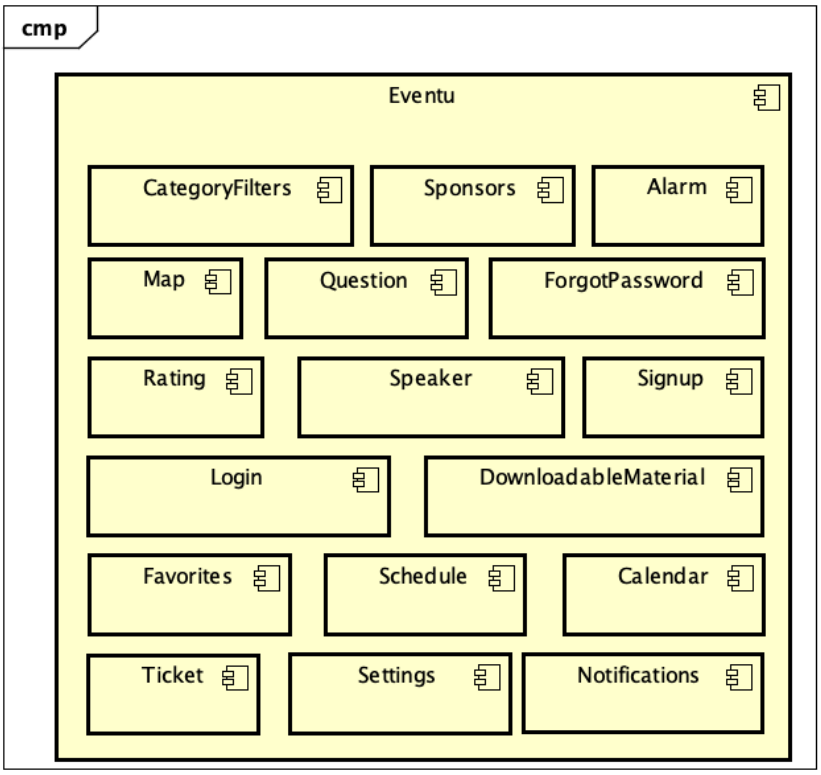

Figura 22 – Estrutura interna Eventu

Para implementação do nosso projeto usamos uma arquitetura mista, aproveitando as melhores características das arquiteturas VIP e VIPER, discutidas na Seção 2.2. Utilizamos como base a VIP, por ser uma arquitetura mais aberta definindo apenas os atores principais e possuir um fluxo unidirecional de dados. A Figura 23 apresenta a arquitetura do projeto.

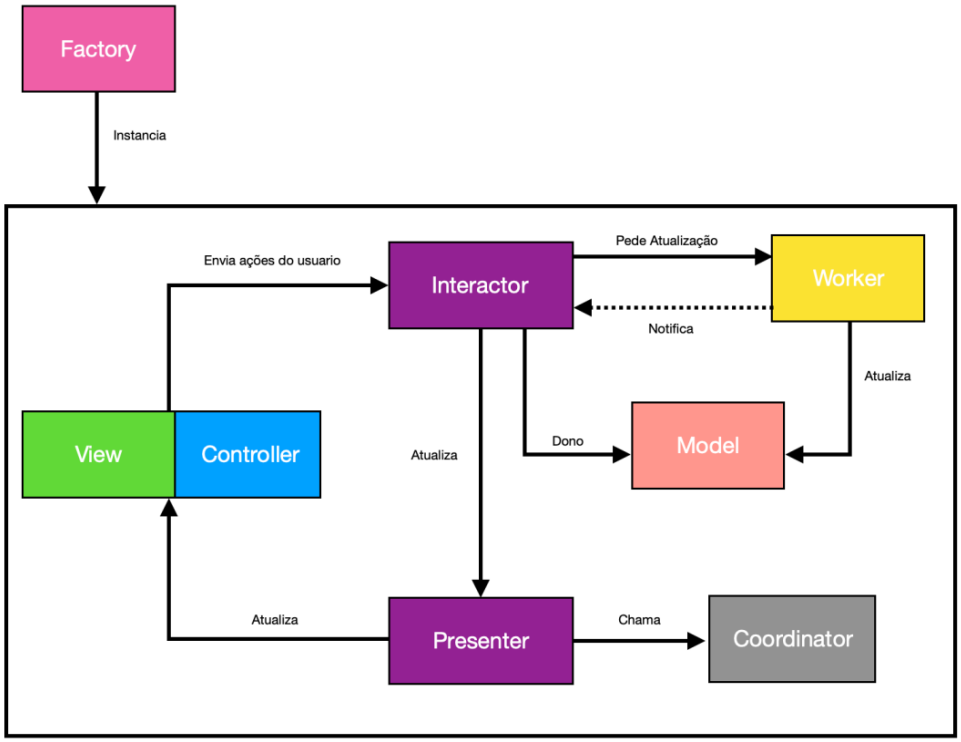

Figura 23 – Arquitetura do Projeto.

Adicionamos um *Worker* para tratar a recuperação de dados da Web ou do banco de dados local, fazendo gerenciamento das *threads* e requisições. O *Coordinator* faz o controle do fluxo, e só pode ser acessado depois de todo um ciclo, onde serão analisadas regras de negócio e apresentação. O *Factory* tem a responsabilidade de instanciar e injetar as dependências necessárias para criar uma nova cena. O *Presenter* cuida da lógica de apresentação e avisa ao *Controller* quando e como ele deve exibir os dados. Essa proposta visa distribuir as responsabilidades e diminuir o acoplamento entre as camadas. Além de se basear no VIPER, a proposta se inspira também no padrão hexagonal.[6]

5.2 Views

A implementação de todas as cenas do *App* é feita utilizando *View Code*, por ter se mostrado uma forma muito mais segura e flexível de desenvolvimento. Quando utilizamos blocos, variáveis e funções de certa forma se torna muito mais fácil criar uma *view* reutilizável e de fácil manutenção. Também podemos perceber o incentivo da Apple na programação de *views* via código com o lançamento do SwiftUI.[7]

Definimos alguns padrões na declaração e composição dos elementos. Na declaração os elementos sempre começam com o modificador de acesso **private**, as atribuições de propriedades só podem ser feitas através de métodos públicos ou variáveis computadas,

[6] <https://java-design-patterns.com/patterns/hexagonal/>
[7] <https://developer.apple.com/xcode/swiftui/>

dessa forma nunca expomos diretamente a implementação da *view*. Em seguida são declarados como **lazy**, pois a inicialização tardia traz algumas vantagens, como a instância só ocorre na primeira vez em que a variável é usada e alocação de memória é feita apenas se necessário e no momento em que for demandada. Também podemos usar o **self**, pelo fato da inicialização acontecer após a execução do construtor da classe. Quanto à composição, ela acontece em 3 momentos. No primeiro usamos a inicialização via *closure* para compor as propriedades do elemento, no segundo montamos toda hierarquia da *view* na chamada da função **addComponents**. E por último no método **layoutComponents** definimos suas *constraints* e prioridades. A Listagem 5.1 exibe o exemplo de uma *view* conforme discutido.

Listagem 5.1 – Exemplo de View Code

```
import UI
import UIKit

class NotificationCellHeader: View {
    private lazy var stackView: UIStackView = {
        let stackView = UIStackView()
        stackView.axis = .vertical
        stackView.alignment = .fill
        stackView.spacing = 3
        stackView.translatesAutoresizingMaskIntoConstraints = false

        return stackView
    }()

    private lazy var nameLabel: UILabel = {
        let label = UILabel()
        label.font = UIFont.systemFont(ofSize: Font.title, weight: .semibold)
        label.textColor = Palette.darkGrey.color

        return label
    }()

    private lazy var informationLabel: UILabel = {
        let label = UILabel()
        label.font = UIFont.systemFont(ofSize: Font.subtitle, weight: .semibold)
        label.textColor = Palette.lightGrey.color

        return label
    }()

    // ....

    override func addComponents() {
        stackView.addArrangedSubview(nameLabel)
        stackView.addArrangedSubview(informationLabel)
        addSubview(stackView)
    }

    override func layoutComponents() {
        NSLayoutConstraint.activate([
            stackView.topAnchor.constraint(equalTo: topAnchor),
            stackView.leadingAnchor.constraint(equalTo: imageView.trailingAnchor,
                constant: Layout.margin),
            stackView.trailingAnchor.constraint(equalTo: trailingAnchor, constant
                : -Layout.margin),
            stackView.bottomAnchor.constraint(equalTo: bottomAnchor)
        ])
    }

    public func configureView(name: String, information: String) {
        nameLabel.text = name
        informationLabel.text = information
    }
    //...
}
```

5.3 Testes

Em nossa experiência, testes são uma das melhores ferramentas disponíveis para a tarefa de programação: ao mesmo tempo que ele consegue eliminar a angústia de saber se o que estamos alterando ou adicionando irá afetar de forma negativa algum pedaço

da nossa aplicação, também nos poupa o árduo trabalho de ficar re-testando a aplicação em diversos contextos atrás de alguma falha de implementação, muitas vezes acabam promovendo o desacoplamento de código, melhoram nossa escrita de código e servem de documentação.

Assim como outras partes do aplicativo, os testes também irão sofrer refatoração, manutenção e adição. Por isso é muito importante implementar padrões de escrita e estruturas (MARTIN, 2009).

Para nomenclatura das funções foi adotado o padrão *Test When Should*: esses três blocos são separados por hífen para facilitar a leitura. O bloco de *Test* diz qual função estamos testando, *When* define em qual contexto ou situação o teste ocorrerá, *Should* descreve o comportamento esperado para o teste. A Listagem 5.2 mostra um exemplo de teste que ilustra esta nomenclatura.

Listagem 5.2 – Exemplo de nomeclatura para teste

```
1 import XCTest
2
3 class SpeakerInteractorTest: XCTestCase {
4     func testUpdateView_WhenDataIsEmpty_ShouldCallUpdateData() {
5         // ...
6         sut.updateView(data: [])
7         // ...
8     }
9 }
```

Para compor a estrutura de teste foram usados os conceitos de *Sut, Mock, Spy*:[8]

- ***Sut (Subject Under Testing)***: representa o objeto que queremos testar;
- ***Mock***: representa a falsificação de uma dependência, retornando um caso de uso conforme necessário, mas obedecendo o contrato pré-definido. Como exemplo não podemos utilizar a camada real de *Service* nos testes, pois deste modo limitamos nossos testes a disponibilidade do servidor, nesse caso criamos um *Mock* desse *Service*, que irá nos trazer retornos pré estabelecidos conforme nossa necessidade;
- ***Spy***: é um objeto que grava suas interações com outros objetos, ele é útil quando precisamos saber se a função que estamos testando chamou alguma outra classe ou quais tipos de argumentos foram passados.

Um exemplo de escritas de teste envolvendo todos esses conceitos é exibida na Listagem 5.3.

[8] <https://martinfowler.com/articles/mocksArentStubs.html>

Listagem 5.3 – Exemplo de atores para teste

```
import XCTest

class SpeakerServiceMock: SpeakerServicing {
    var isEmpty: Bool = false

    func getAllSpeakers(completion: @escaping(Result<[Speaker], EventuError>) ->
        Void) {
        guard isEmpty else {
            //...
            return
        }

        completion(.success([]))
    }

    //...
}

class SpeakerPresenterSpy: SpeakerPresenting {
    private(set) var callUpdateDataCount: Int = 0
    private(set) var data: [Speaker]?

    func updateData(model: [Speaker]) {
        callPerformActionCount += 1
        self.data = model
    }

    //...
}

class SpeakerInteractorTest: XCTestCase {
    private let presenterSpy = SpeakerPresenterSpy()
    private let serviceMock = SpeakerServiceMock()

    private lazy var sut: SpeakerInteractor = {
        let sut = SpeakerInteractorinit(service: serviceMock, presenter:
            presenterSpy)
        return sut
    }()

    func testUpdateView_WhenDataIsEmpty_ShouldCallUpdateData() {
        serviceMock.isEmpty = true

        sut.updateView()

        XCTAssertEqual(presenterSpy.callUpdateDataCount, 1)
        XCTAssertTrue(presenterSpy.data.isEmpty)
    }
}
```

Para o nosso projeto implementamos testes unitários e de interface. Os testes unitários foram implementados nas camadas *Interactor* e *Presenter* pois, devido à arquitetura proposta, essas classes concentram toda lógica responsável por uma cena.

Os testes de interface, por serem muitos custosos, foram implementadas em fluxos essenciais da aplicação, sendo nossa principal preocupação nesse ponto testar se o usuário consegue fazer uma determinada navegação. Aqui buscamos fugir de testes que checam tamanho, fonte e cor de componentes, para esse caso seria mais conveniente fazer um teste de *SnapShot*. Foi escolhido o teste de navegação pois ao focarmos somente em fluxos importantes, economizamos tempo de desenvolvimento e *CI*, porque não testamos o

comportamento de todas as funcionalidades do Aplicativo.

Todos os testes foram implementados usando a biblioteca nativa *XCTest*.

5.4 Firebase

Firebase é uma plataforma do Google que oferece vários serviços que auxiliam na criação e escalabilidade de *Apps mobile*, essa seção se destina a mostrar quais serviços do Firebase foram utilizados.

5.4.1 *Cloud Storage*

É um serviço de armazenamento de alta performance e escalabilidade, usado para armazenar conteúdos gerados pelo usuário, como áudio, vídeo e documentos. Para o projeto Eventu, ele é utilizado para o armazenamento de imagens do usuário. O usuário pode definir uma imagem em 3 pontos do *App*, conforme ilustrado na Figura 24:

- Ao se cadastrar via e-mail;
- Editar a foto nas configurações;
- Login via Google, neste caso é usada a imagem definida pelo usuário nos serviços do Google.

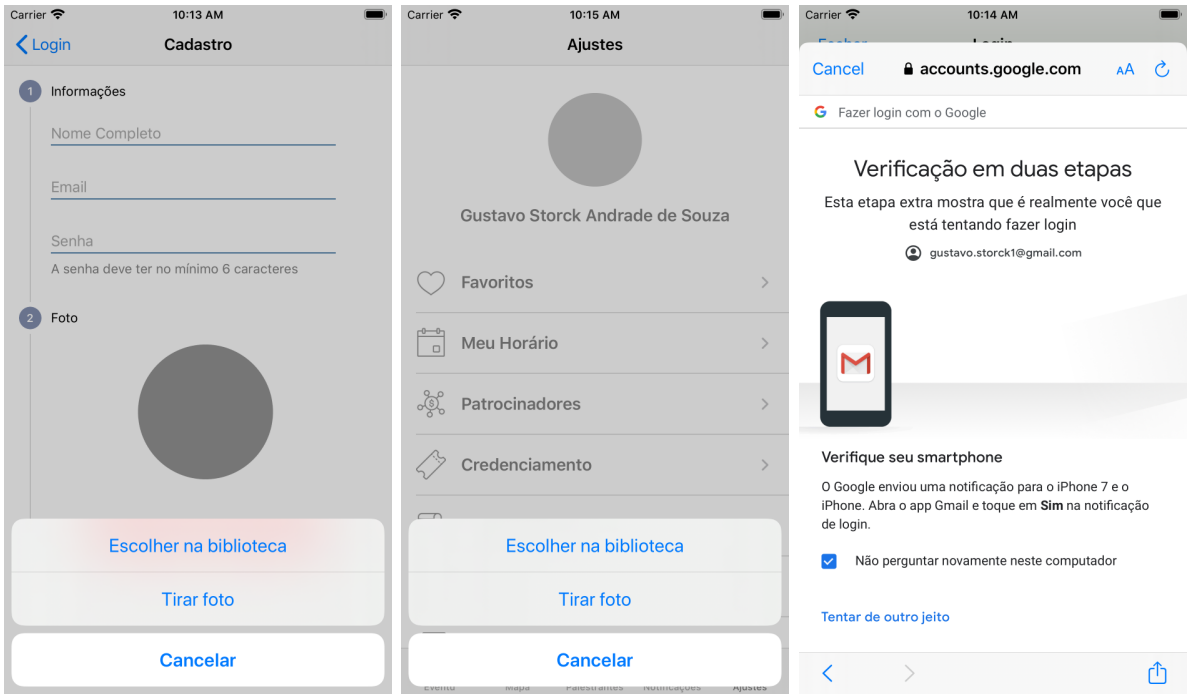

Figura 24 – Upload de imagem

Um dos maiores benefícios do *Cloud Storage*[9] é cuidar inteiramente do *upload* da imagem, tratando erros como, por exemplo, se a conexão com a Internet cair no meio do processo de *upload*, a biblioteca consegue iniciar o *download* do último passo válido. Além disso é criada uma representação das imagens no console do Firebase, ilustrado na Figura 25, onde é possível fazer algumas edições, acompanhar o tráfego, e definir regras de segurança.

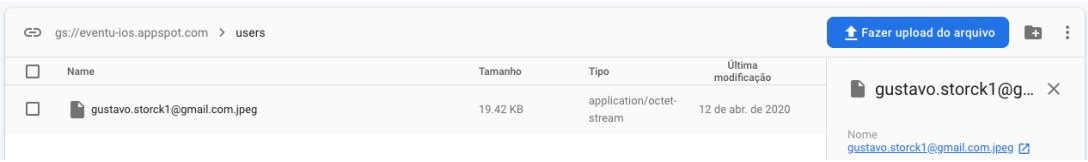

Figura 25 – Painel Cloud Storage

Fluxo de *upload*: ao solicitar o processo de *upload*, o aplicativo comprime a imagem, cria uma URL única baseada no e-mail do usuário e chama a biblioteca passando a URL e a imagem comprimida. Ao concluir o processo de *upload*, uma URL final é retornada referenciando o local onde a imagem está armazenada e essa URL é salva no modelo do usuário para ser usada em outros pontos do *App*.

5.4.2 *Analytics*

É uma solução de análise de *Apps* que fornece informações sobre o uso do aplicativo e o envolvimento do usuário. O *SDK* de *Analytics*[10] captura automaticamente uma série de eventos e propriedades do usuário, como tipo de celular, sexo, região, idade, quantidade de usuários online, retenção do usuário, além de permitir que você defina eventos personalizados.

Isso permite traçarmos um perfil muito mais detalhado sobre o público do cliente, além de nos ajudar a entender como está a aceitação do *App* para os usuários. O *App* implementa eventos para cada tela que foi aberta e se o usuário conseguiu fazer determinada ação, como enviar uma pergunta ou fazer uma avaliação. A Figura 26 mostra a lista de eventos personalizados para o aplicativo desenvolvido.

[9] <https://firebase.google.com/docs/storage>
[10] <https://firebase.google.com/docs/analytics>

Eventos existentes

Nome do evento	Contagem	% de variação	Usuários	% de variação	Marcar como conversão
Abriu_Ajuda	2	-	2	-	
Abriu_Ajustes	39	-	7	-	
Abriu_Avaliações	3	-	1	-	
Abriu_Cadastrar	1	-	1	-	
Abriu_Calendario	1	-	1	-	
Abriu_Downloads	1	-	1	-	
Abriu_Filtros	3	-	2	-	
Abriu_Inicio	130	-	18	-	
Abriu_Login	7	-	4	-	
Abriu_Mapas	6	-	5	-	

Figura 26 – Eventos personalizados

Esse tipo de métrica nos ajuda a melhorar, criar, excluir, reorganizar funcionalidades, de forma que suas funções sejam mais intuitivas e relevantes. Podemos testar funcionalidades em determinado segmento, de região, idade, etc. É muito importante tomarmos decisões baseadas em dados, pois dessa forma nos tornamos mais assertivos.

Os dados capturados ficam disponíveis em um painel no Console do Firebase, como mostra a Figura 27.

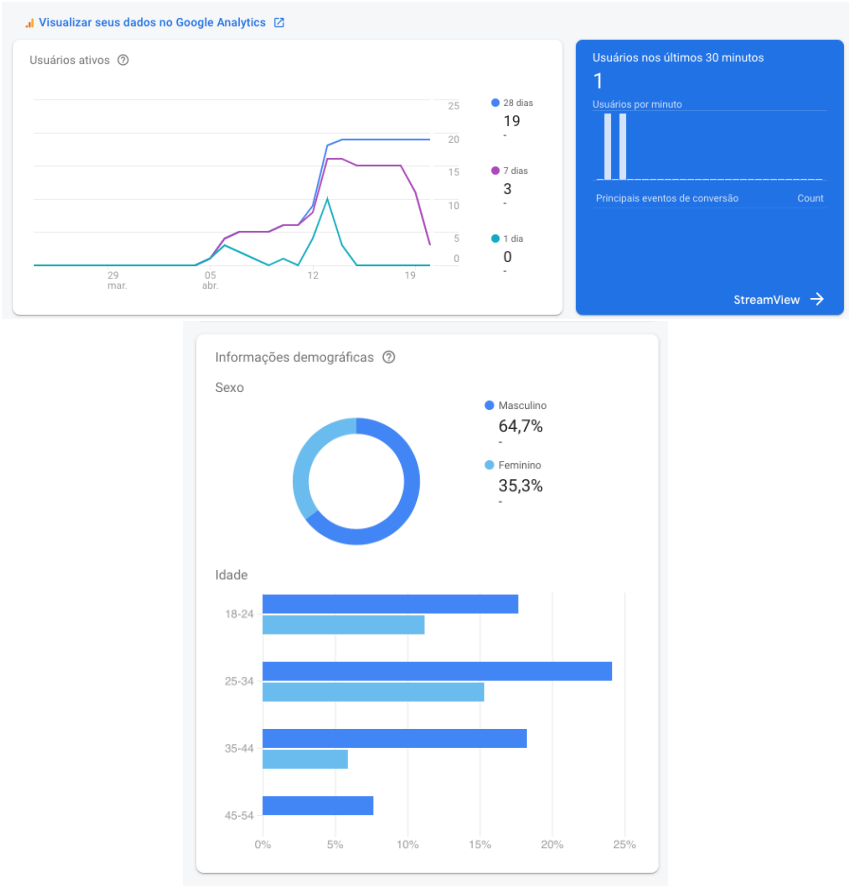

Figura 27 – Painel de dados

5.4.3 *Crashlytics*

O *Crashlytics*[11] é uma ferramenta de relatório de falhas em tempo real que ajuda a monitorar, priorizar e corrigir problemas de estabilidade que comprometem a qualidade do seu aplicativo. Após sua integração com o App, é possível obter em detalhes informações sobre alguma falha, como versão do dispositivo, versão do sistema operacional, versão do aplicativo e um *stack trace* detalhado. Com isso em mãos é muito mais fácil priorizar e reproduzir um problema. Foi uma excelente solução ao problema que ocorria há alguns anos, quando não conseguimos saber se usuários estavam sendo afetados por algum *bug*, pois muitas vezes quando o problema era relatado de maneira manual era muito difícil saber sua origem. A Figura 28 mostra o painel do *Crashlytics*.

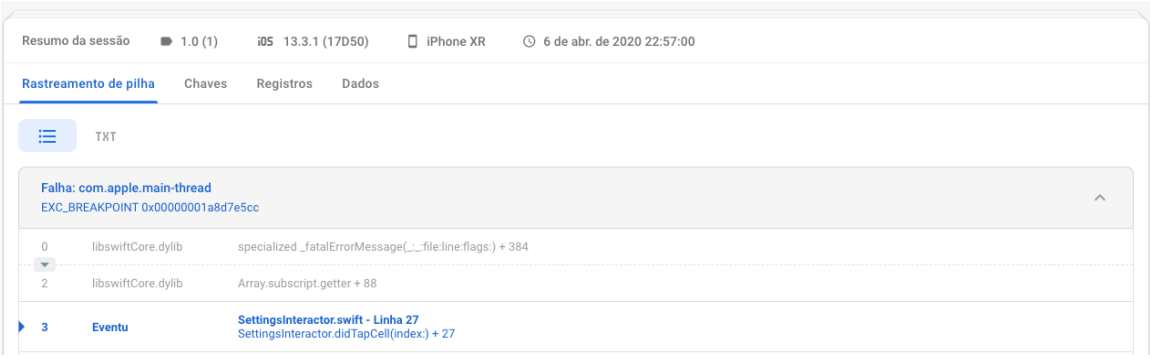

Figura 28 – Painel Crashlytics

5.4.4 *Authentication*

Autenticação é um ponto muito delicado da maioria das aplicações, por questões de segurança tende a ser muito difícil e custosa de ser implementada do zero. O Firebase *Authentication*[12] resolve nosso problema, fornecendo serviços de *back-end, SDKs* prontas para autenticar usuários no *App*. Com suporte à autenticação por meio de senhas, números de telefone e provedores como Google e Facebook. Login por provedores acabam sendo mais escolhidos pelos usuários devido à facilidade e velocidade.

Na implementação do Eventu, escolhemos a autenticação via e-mail/senha e pelo Google. O *App* usa a biblioteca *Authentication* para enviar um *token* único por usuários em rotas que devem ser autenticadas, como atualizar perfil, enviar pergunta, fazer avaliação, etc. Nessas requisições é adicionado ao *header* um campo *Authorization*, que contém o *token* válido por seção. Por questões de segurança, esse *token* é renovado a cada 5 minutos.

A biblioteca também possui suporte a cadastro. Por requisito de sistema, podemos ter dois tipos de cadastro: o primeiro é quando pode ser feito pelo *App* utilizando e-mail/senha ou o provedor do Google, já o segundo ocorre quando o cadastro deve ser

[11] <https://firebase.google.com/docs/crashlytics>
[12] <https://firebase.google.com/docs/auth>

controlado por uma página Web e, nesse caso, apenas o login é feito pelo aplicativo, enquanto o botão de cadastro direciona para a página Web correspondente. A Figura 29 ilustra os diferentes métodos de login no aplicativo.

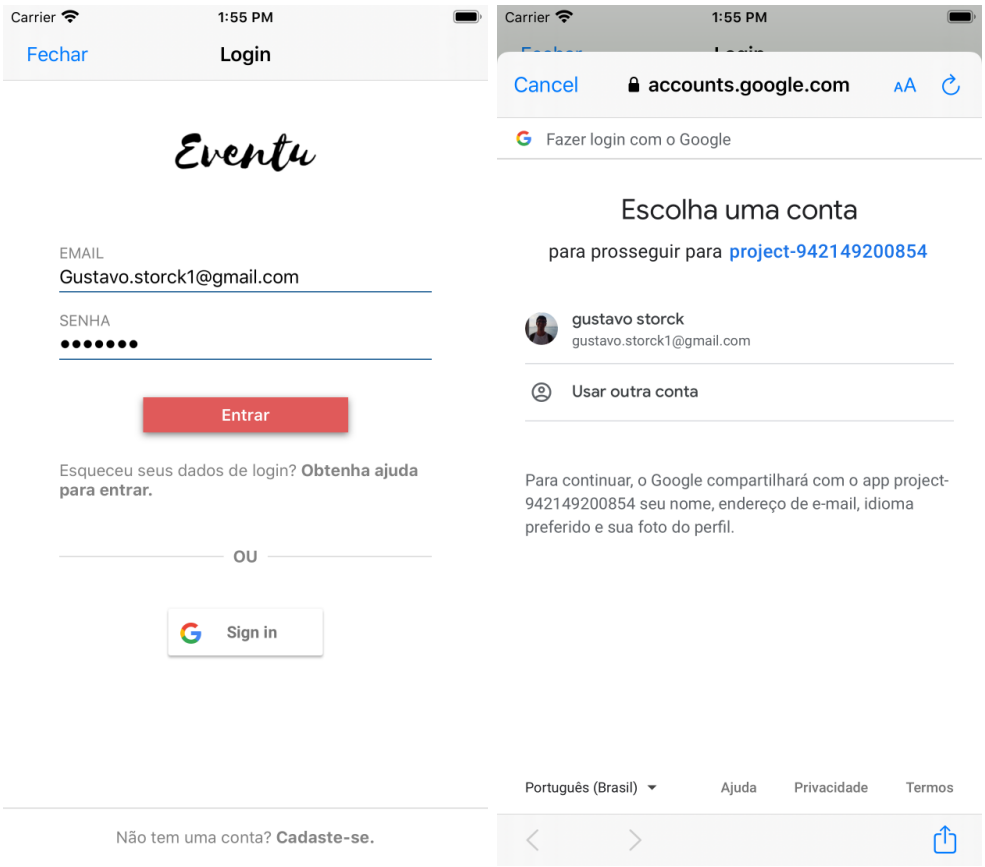

Figura 29 – Métodos de login

A biblioteca de *Authentication*, também implementa métodos redefinir/recuperar senha, através do e-mail. Além de possuir um painel para configurações da funcionalidade, como mostra a Figura 30.

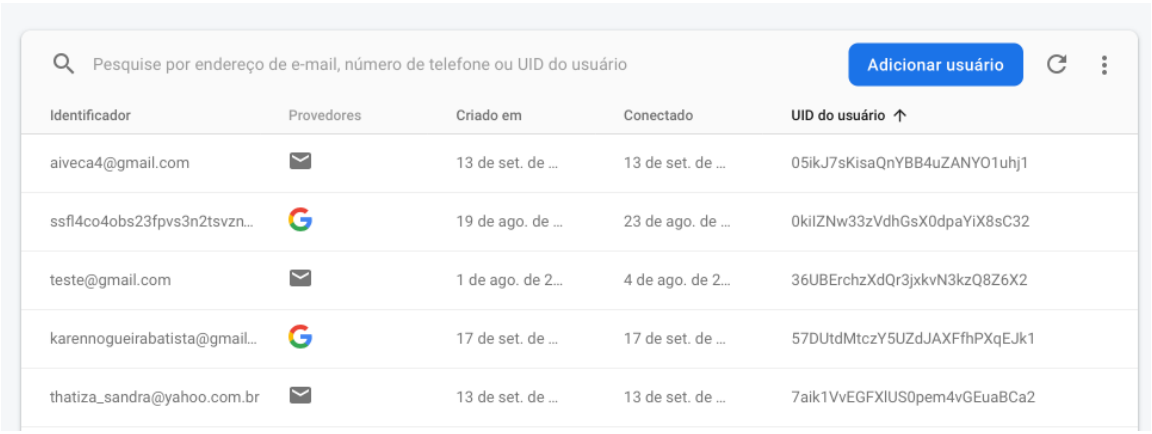

Figura 30 – Painel com login dos usuários

5.4.5 *Remote Config*

Remote Config[13] é um serviço em nuvem que permite alterar um comportamento ou aparência do *App* sem exigir que os usuários façam o *download* de uma atualização. Isso abre diversas possibilidades, nos permite fazer teste A/B[14] a um determinado segmento de usuários, podemos desabilitar um fluxo que não esteja funcionando corretamente, etc.

No entanto, a principal vantagem para nós foi a versatilidade: na concepção do aplicativo definimos que ele tinha que ser customizável para atender a público e demandas diferentes, sem que exigisse muito esforço. O *Remote Config* se encaixa perfeitamente como uma das formas de alcançar isso: o *App* foi programado para disponibilizar e se comportar de acordo com algumas *flags*, que podem ser habilitadas por evento de forma remota, como mostra a Figura 31. Então, por exemplo, se o cliente não quiser o fluxo de login no *App*, basta desabilitar em um arquivo de configuração e pronto, não temos que mexer em uma linha de código. Além de velocidade, nos trás segurança, pois como esses comportamentos já são esperados podemos escrever testes, evitando que alguém mexa com pressa no código e acabe introduzindo um *bug*.

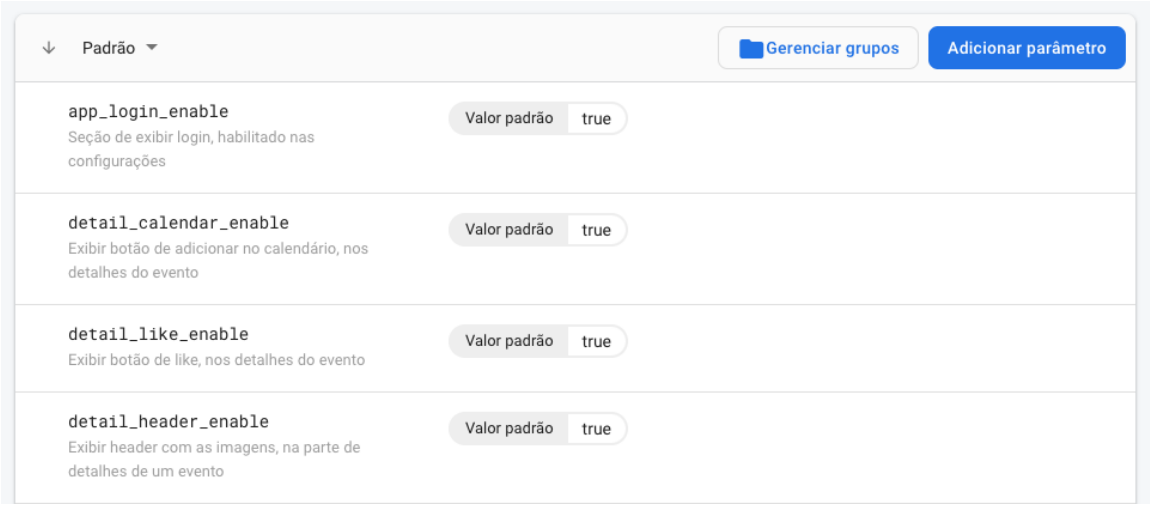

Figura 31 – Painel com as Flags

Através do *Remote Config* também é possível organizar uma implantação que seja gradual e possua versionamento da base, podemos habilitar um fluxo ou funcionalidade, levando em consideração versão do aplicativo, sistema operacional, e porcentagem de usuários que desejamos atingir.

No entanto, como nem tudo é perfeito, o Google avisa que uma mudança pode demorar até 12 horas para refletir nos usuários, portanto temos que ficar muito atentos ao mexer nesse arquivo, pois 12 horas podem chegar a ser até um quarto da duração de um evento.

[13] <https://firebase.google.com/docs/remote-config>
[14] <https://firebase.google.com/docs/ab-testing>

5.4.6 *Cloud Messaging*

Cloud Messaging[15] é uma solução para envio confiável de notificações entre plataformas, podendo ser usada de 2 maneiras. A primeira como *Push Notification* Silencioso, por meio da qual o servidor avisa para o aplicativo que existe algum dado disponível para atualização e esta pode fazer a busca e a atualização destes dados em *background*. A segunda forma é para o envio de mensagens, então o sistema da Eventu através do *Cloud Messaging*, permite que administradores do evento notifiquem usuários para promover novas interações e a retenção dos mesmos, como mostrado na Figura 32.

Figura 32 – Exemplo de Push Notification

O serviço de *Push Notification* fornecido pela Apple é o *APNs (Apple Push Notification service)*. Como forma de centralizar o serviço, facilitando o envio tanto para iOS quanto Android, utilizamos o *provider* fornecido pela Google que, ao identificar para qual plataforma ele deve enviar o *push*, chama o serviço específico.

[15] <https://firebase.google.com/docs/cloud-messaging>

5.5 Apresentação do sistema

Nesta seção, será apresentado o sistema por meio de uma série de capturas de tela. A Figura 33 mostra ícone do aplicativo e a primeira tela de inicialização. Esses dois conteúdos são personalizáveis através do esquema de *targets*, cada evento possui seu próprio *target*.

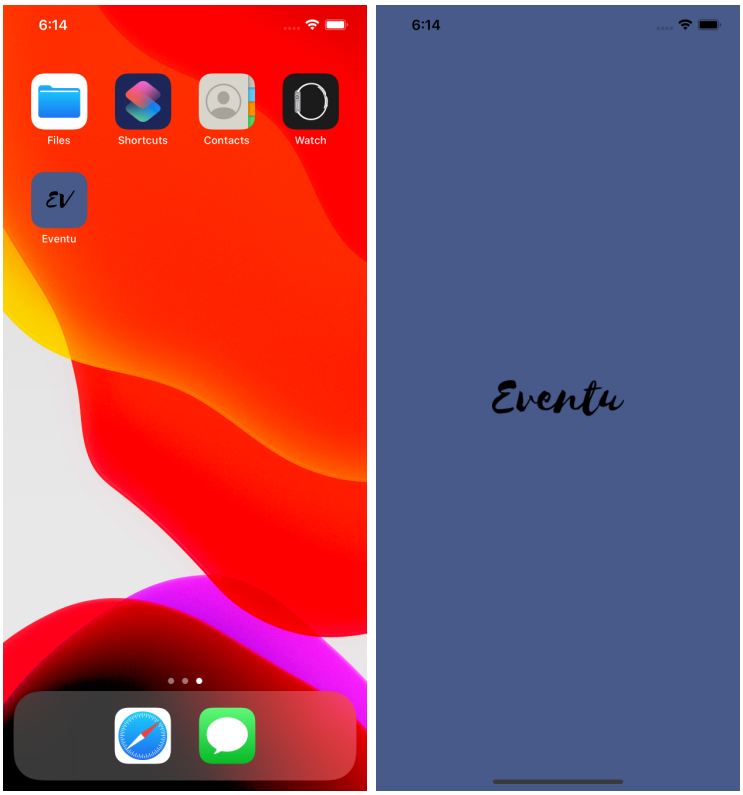

Figura 33 – Inicialização do App

A navegação principal do App é feita através da *TabBar*. A Figura 34 apresenta a seção inicial, ela é descrita com o nome do evento, exibe todas atividades que ocorrerão no dia selecionado na aba superior, ordenadas por hora de início. Fazem parte da interface título, local, horário e cor, sendo que as cores servem para criar categorias visuais das atividades.

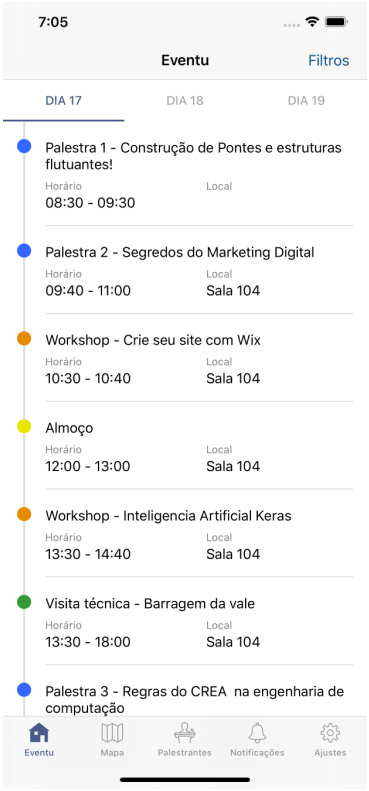

Figura 34 – *Home*

Ao clicar em uma atividade na *home* é exibida uma tela de detalhes, customizável, sendo responsabilidade do *backend* exibir e preencher as seções. As únicas que obrigatoriamente devem aparecer são título e horário. A Figura 35 apresenta a tela de detalhes, no topo estão localizados os botões de ação para favoritar e adicionar na agenda, sua visibilidade é condicionada pelo *remote config* e pelo *response* do *backend*, também armazenam a última opção selecionada. A tela pode possuir um *header* com *scroll* vertical de imagens. A primeira seção contém o título e descrição do evento. A segunda é destinada para exibir os palestrantes do evento. Na seção de horário é possível definir um lembrete para atividade. Depois temos uma seção de informações genéricas com título e descrição. A quinta seção traz informações gerais como duração pré requisitos e idioma da atividade. Por último temos as seções de interação com usuário.

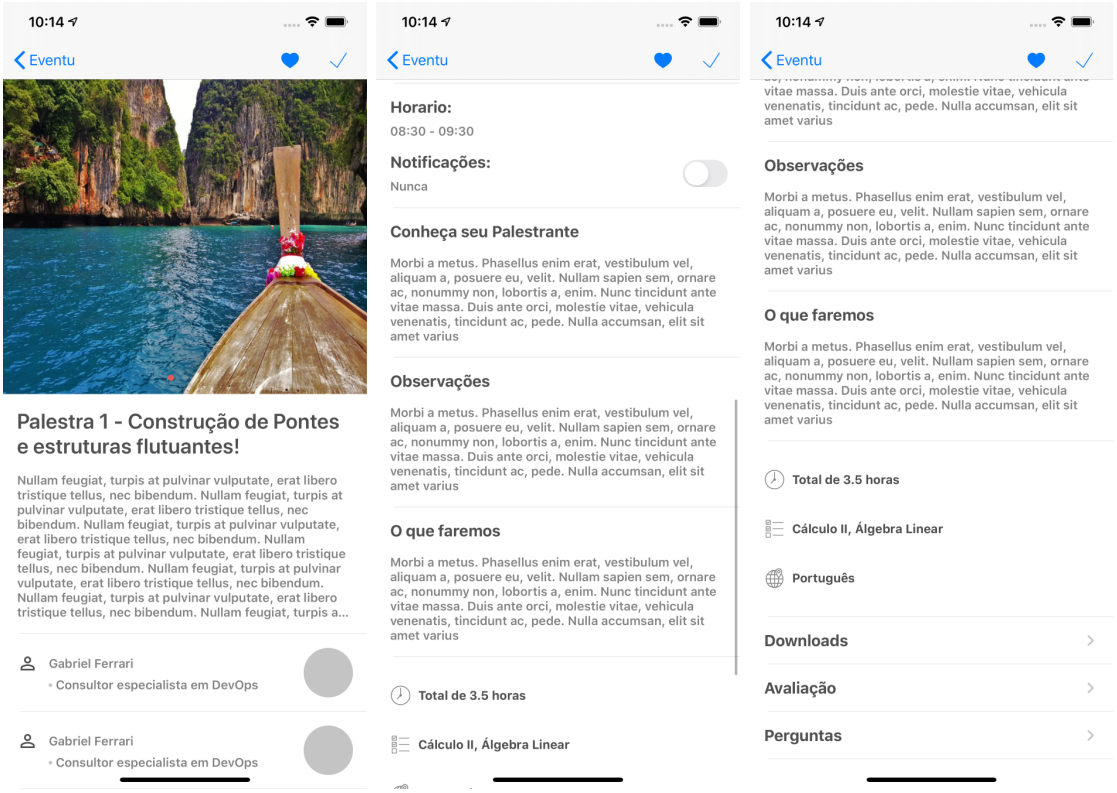

Figura 35 – Detalhes Atividade

Avaliação e Perguntas dependem que o usuário esteja autenticado para prosseguir, caso contrário é sugerido o fluxo de login, através de um alerta, conforme mostrado na Figura 36.

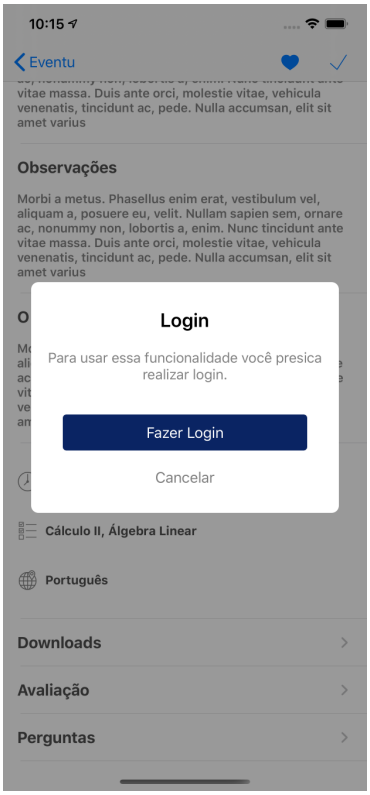

Figura 36 – Sugestão Login

Os 3 itens exibidos na Figura 37, representam interações diretas com usuário seja fazendo uma pergunta, avaliando a atividade ou fazendo o download do material.

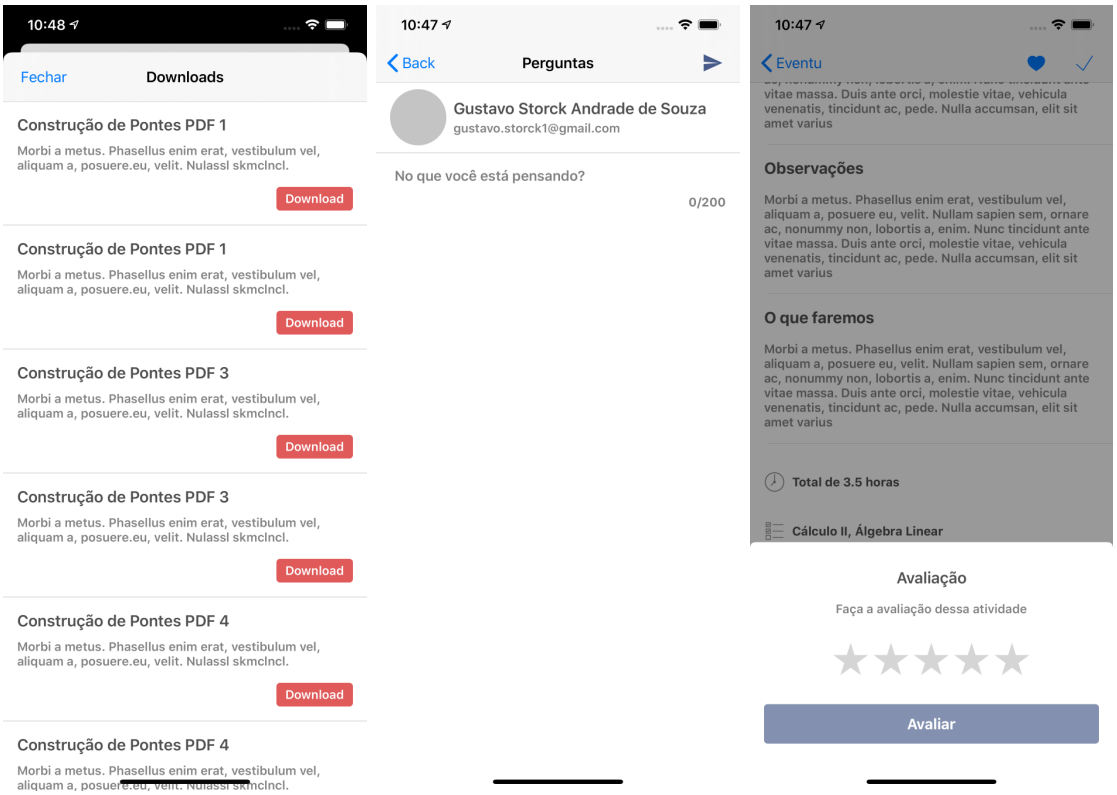

Figura 37 – Interações com o usuário

Os filtros por categorias de atividade, mostrado na Figura 38, são acessados através da *home*, ao selecionar e aplicar um conjunto de itens, a lista principal de atividades é modificada, também é exibido a quantidade de categorias que foram escolhidas à direita do botão Filtros.

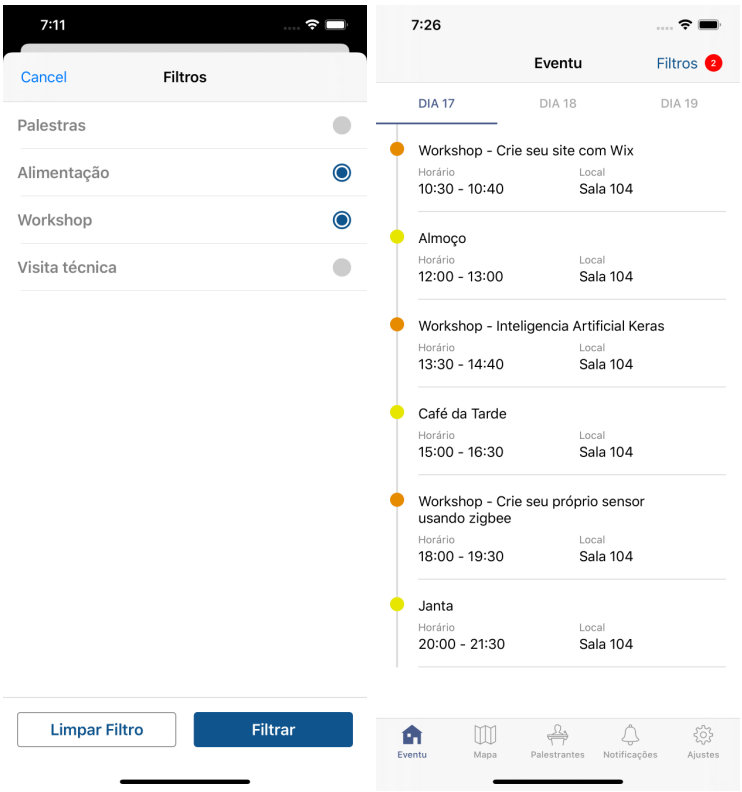

Figura 38 – Filtro de atividade

A Figura 39 apresenta a aba Mapa, responsável por listar todas as localidades do evento. Possui uma série de filtros, sendo possível também pesquisar por nome, alterando a lista interna de locais. Ao selecionar um item da lista interna o mapa se move até a região desejada. O filtro de categoria, com *scroll* vertical, altera a visualização externa do mapa e a lista interna.

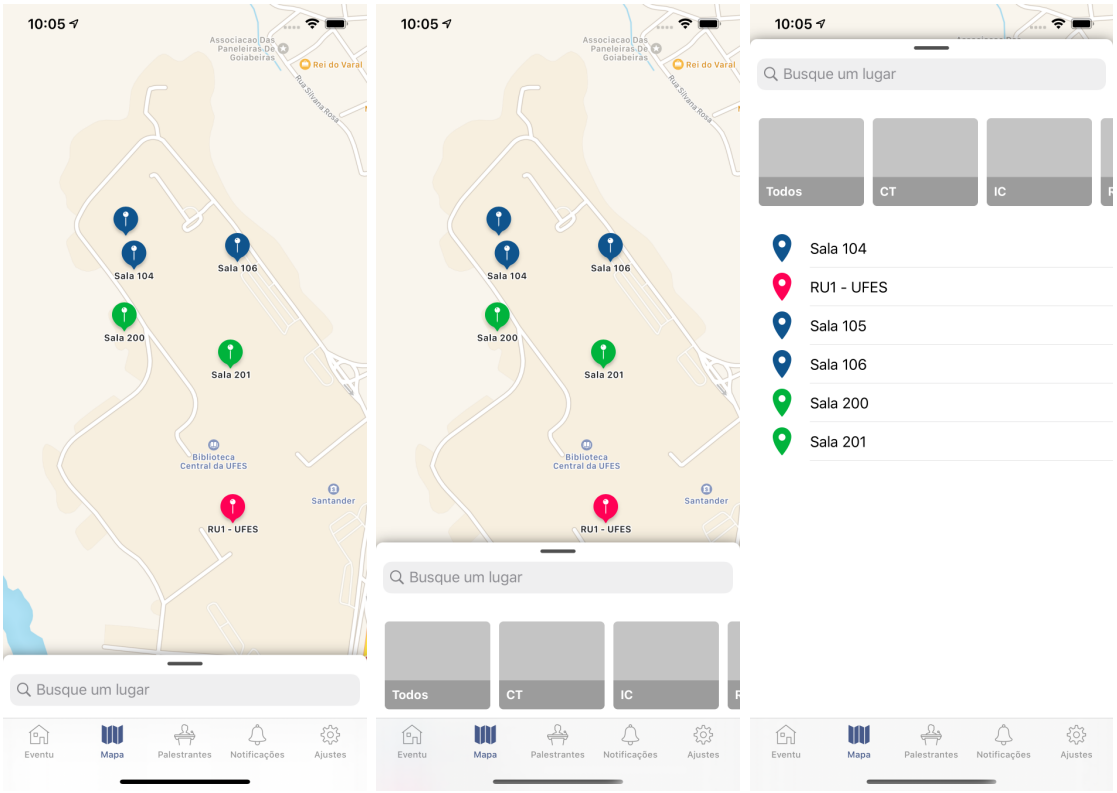

Figura 39 – Mapa

A Figura 40 representa a aba Palestrantes, onde são listados todos os palestrantes do evento, incluindo nome, foto, área de trabalho. Ao clicar em um item, uma descrição mais detalhada é fornecida.

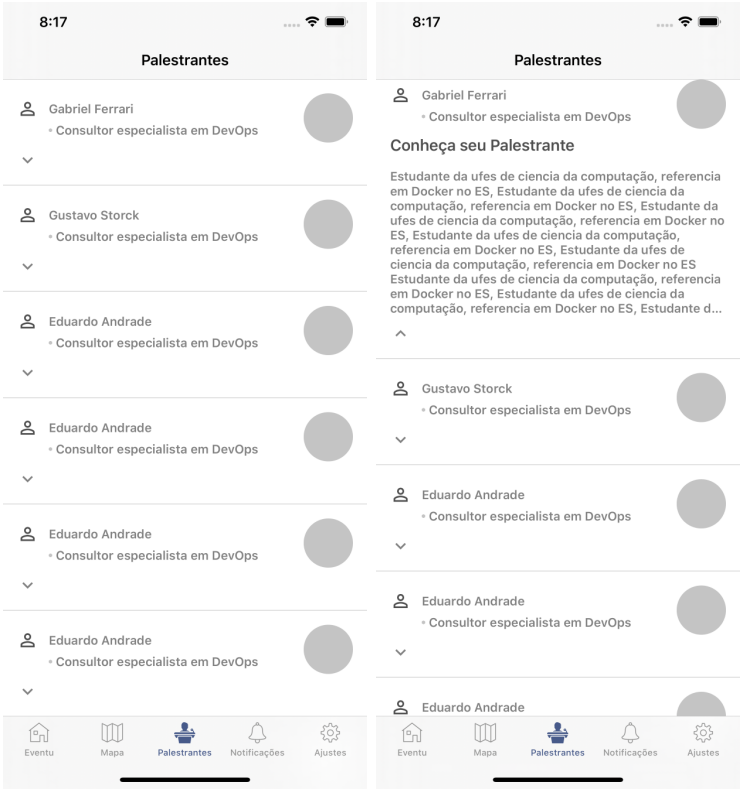

Figura 40 – Palestrantes

A Figura 41 exibe a aba de Notificações, é uma área destinada aos administradores do evento para publicar novidades, informações, avisos. Podem existir 3 tipos de configurações de mensagem: texto mais imagem, somente imagem, somente texto. É uma lista paginada, conforme o *scroll* chega ao fim uma nova requisição é feita pedindo mais itens.

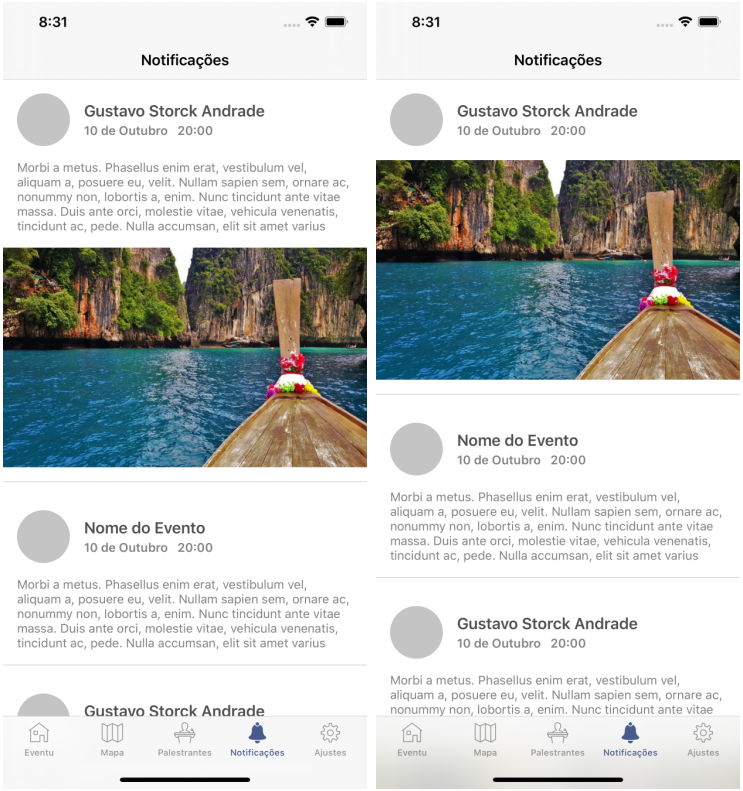

Figura 41 – Notificações

A aba de Ajustes, conforme mostra a Figura 42, provê a entrada para diversos outros fluxos, além de exercer o controle sobre o perfil do usuário, algumas seções como Sair, Credenciamento, Informações sobre a conta, só aparecem caso o login tenha sido realizado. Se o usuário não tiver se identificado aparecerá uma seção para o login.

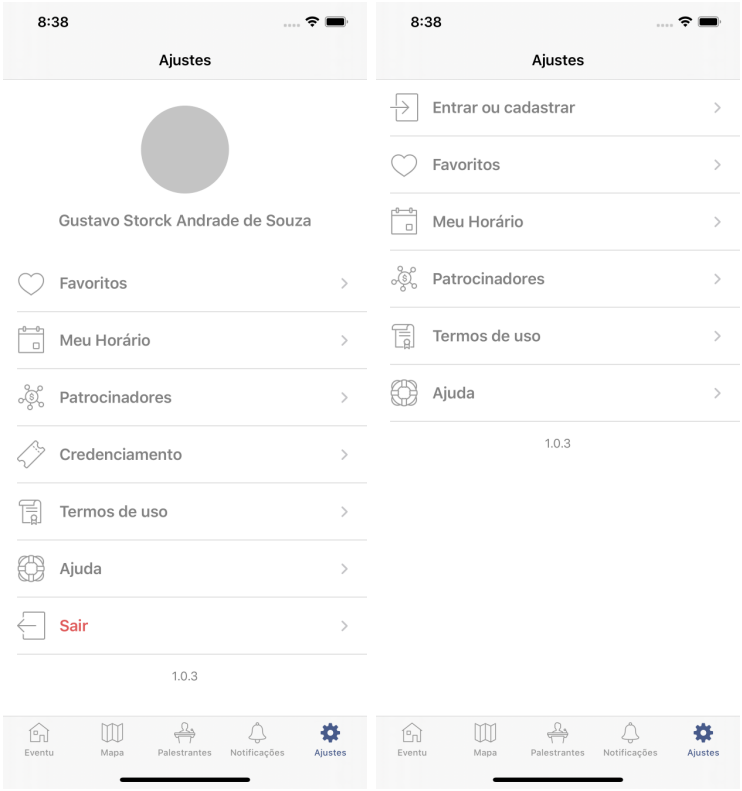

Figura 42 – Ajustes

A Figura 43 apresenta a tela de login, responsável pela autenticação do usuário via e-mail e senha ou pelo provedor do Google, além de ter a opção recuperar senha e entrada para o fluxo de cadastro.

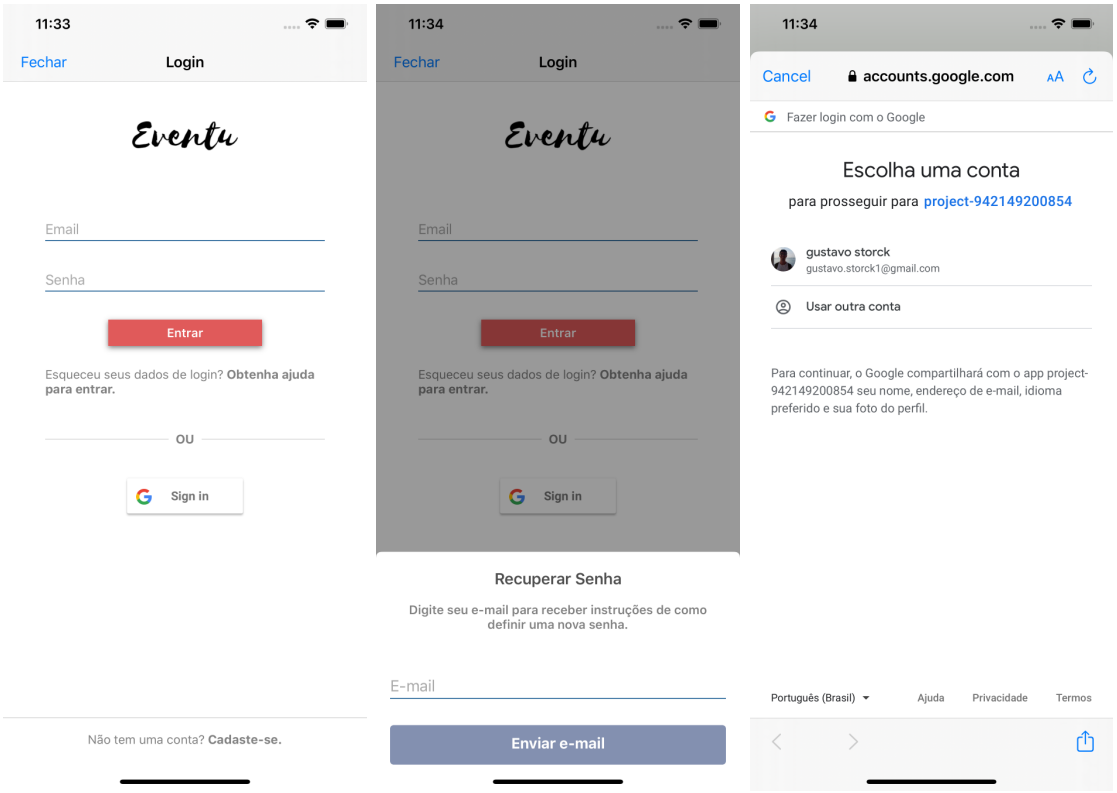

Figura 43 – Login

O cliente possui a opção de permitir ou não o cadastro de um usuário pelo App, caso o cadastro esteja ativo ele pode ser feito via provedor do Google ou inserindo os dados, como demonstrado na Figura 44. Se estiver desabilitado ao clicar em se cadastrar na tela de login ele será direcionado para um site definido pelo cliente.

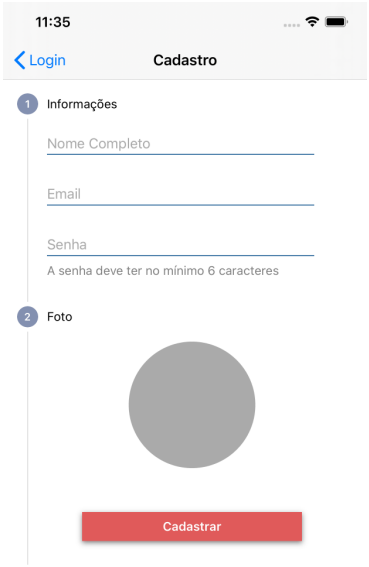

Figura 44 – Cadastro

A Figura 45 exibe as tela de Favoritos e Meu Horário, ambas listam atividades adicionadas nesses tipos específicos pela tela de descrição de atividade, apresentada na Figura 35. O Meu Horário também exibe atividades cadastradas pelo usuário no site do evento. Ambas as seções têm a visibilidade controladas por *remote config* e aparecerem mesmo sem efetuar o login. Ao clicar em um item é aberta a tela de detalhes da atividade.

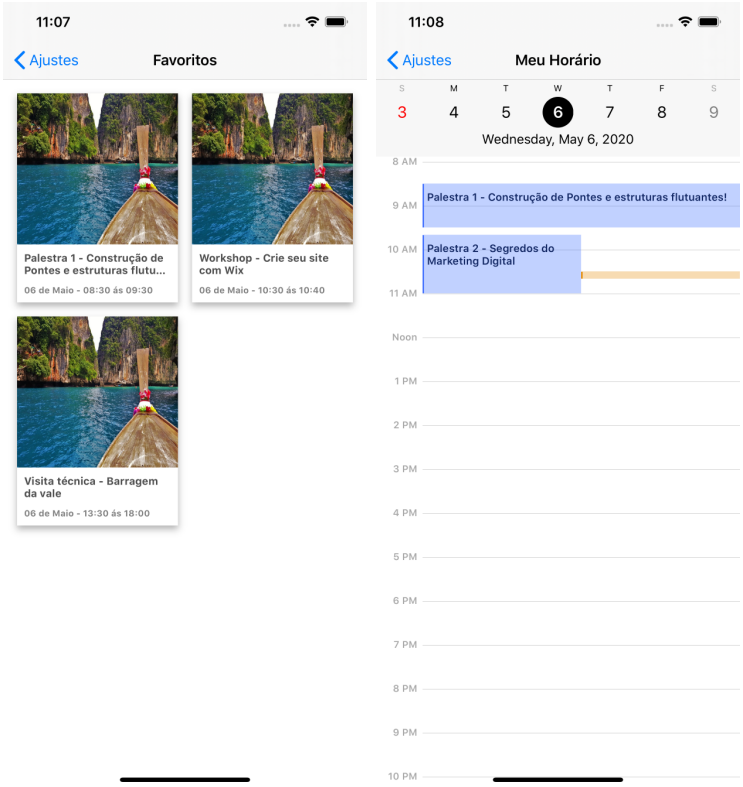

Figura 45 – Favoritos e Meu Horário

A tela de Patrocinadores, exibida na Figura 42, lista os patrocinadores do evento divididos por cotas ou categorias. Possuem a visibilidade controlada pelo *Remote Config*.

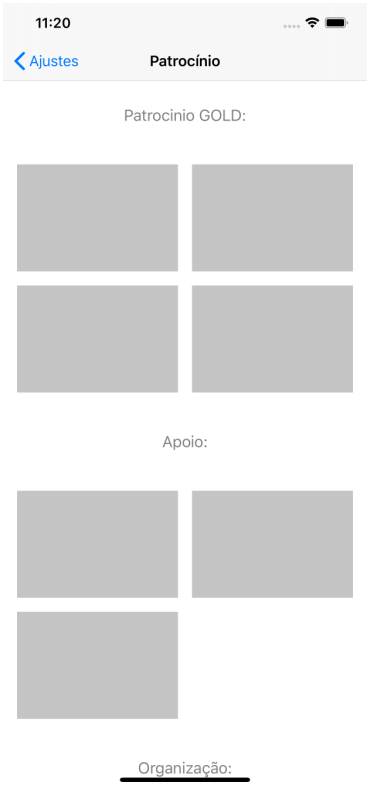

Figura 46 – Patrocinadores

A Figura 47 mostra a tela de Credenciamento, que só é exibida caso o usuário esteja autenticado, cadastrado no evento e o *remote config* habilitado. Ele informa sobre o número do ingresso e exibe um QRCode deste número, para controle e gerência interna do evento.

Figura 47 – Credenciamento

Por fim, Termos de Uso e Ajuda direcionam o usuário para páginas Web com conteúdos específico fornecidos pelo cliente.

6 Considerações Finais

Este capítulo apresenta as conclusões do trabalho realizado. A Seção 6.1 relata os resultados obtidos e o crescimento pessoal. Na Seção 6.2 relatamos como foi a experiência de aplicar o produto no evento CONBEA 2019. Por fim, a Seção 6.3 apresenta as limitações e perspectivas de trabalhos futuros.

6.1 Conclusões

Todos objetivos levantados no Capítulo 1 foram alcançados. O aplicativo foi utilizado no segundo semestre de 2019, como uma forma de informação e interação, trazendo detalhes sobre palestras, atividades, notícias, votação, fórum de perguntas, para os participantes dos eventos CONBEA 2019 e da Semana de Engenharia, SENG 2019. Foi possível perceber a nítida economia de tempo: por exemplo, para criação de um *target* e *remote config* foram cerca de 30 minutos, sendo que anteriormente gastávamos 2 horas e meia. Os números de *bugs* encontrados foram bem inferiores em relação ao aplicativo Android, desenvolvido em outra época sem serem aplicados os métodos discutidos neste trabalho, chegando a ser zero no iOS em um dos eventos. Os *bugs* encontrados podiam ser facilmente identificados e corrigidos com o uso do *Crashlytics*. Os usuários se mostraram muito satisfeitos com o aplicativo iOS. Conseguimos dados para avaliar o comportamento de uso e o perfil dos nossos usuários. Por fim, em ambos eventos não precisamos subir nenhum *update* de ajuste.

Parte dessas conquistas foi devido à programação aplicando *Clean Code*, a arquitetura baseada no *Clean Architecture* e ao uso de testes. O projeto demorou bem mais a ser desenvolvido em relação ao Android, mas em contrapartida é muito mais fácil adicionar uma nova funcionalidade, um novo cliente e garantir a qualidade final do produto, o que se traduziu em economia de tempo a longo prazo. Hoje eu consigo imaginar um time de várias pessoas trabalhando no mesmo App sem prejuízo de produtividade e qualidade, por conta dos processos de apoio ao desenvolvimento.

Toda essa experiência contribuiu muito para o meu desenvolvimento pessoal, a graduação me deu uma base sólida para que eu continuasse aprendendo e trilhasse novos caminhos sempre com pensamento crítico. Há 1 ano e 6 meses eu nunca tinha desenvolvido uma linha de código em iOS, não sabia criar um teste unitário e não conhecia um único *design pattern*. Foram longos meses que abriram minha mente para os benefícios e desafios adotando-se práticas visando à qualidade do produto. Esse trabalho foi o início de uma longa e excitante jornada na busca de mais conhecimento. De certa forma, me parece que só aprendemos a programar errando, ajustando e programando. Apenas a prática leva à perfeição.

6.2 CONBEA - 2019

O Congresso Brasileiro de Engenharia Agrícola (CONBEA) é um dos eventos mais tradicionais no Brasil, mantido anualmente pela Associação Brasileira de Engenharia Agrícola (SBEA). O CONBEA 2019, em sua 48ª edição, foi realizado em Campinas - SP, no período de 17 a 19 de setembro de 2019. Ao total foram 357 *downloads* do aplicativo, sendo 80% Android e 20% iOS. Destes, 64,7% eram do sexo masculino e 35,3% feminino. A maior parte do público, cerca de 40% tinham entre 25 à 34 anos. Todos esses dados estatísticos foram coletados devido aos *logs* implementado no App. Além disso, também conseguimos extrair algumas informações sobre as nossas funcionalidades, como por exemplo: dos 72 usuários iOS, apenas 18 fizeram a avaliação de alguma atividade, isso nos ajuda a tomar medidas para melhorar a exposição de funcionalidades e termos uma comunicação mais clara para os organizadores sobre como orientar os usuários a usar determinadas funções.

O fluxo de trabalho foi feito da seguinte forma: após o fechamento do contrato, enviamos um formulário, com perguntas sobre as configurações do *app*, como cor, nome, quantidade de dias, logo e quais funcionalidades estariam inclusas. Também enviamos chave de acesso para os administrados, para que esses possam cadastrar as atividades locais e demais informações. Após a resposta, iniciamos a configuração do aplicativo. Para o iOS, devido aos métodos aplicados neste trabalho, este processo levou poucas horas. Após concluída essa etapa, liberamos em versão beta o *app* para os administradores. Quando tudo estava concluído, o cliente escolheu a data para publicarmos os aplicativos. Durante o evento foi feito o acompanhamento das funcionalidades e da estabilidade. Ao final, foi gerado um relatório sobre os usuários e falhas coletadas.

6.3 Limitações e Perspectivas Futuras

A principal limitação ao decorrer do desenvolvimento foi o tamanho da equipe: todo aplicativo iOS foi desenvolvido por uma única pessoa. Isso acaba impedindo que decisões sejam tomadas a partir de discussão e levando em consideração diferentes opiniões e pontos de vista. Algumas práticas realizadas visando à qualidade do produto, como *code review* e testes exploratórios das funcionalidades, também foram afetadas.

O desenvolvimento de software pode ser considerado muito volátil, tanto em termos de tecnologia quanto de requisitos, por isso não conseguimos explorar todo potencial somente neste trabalho. Assim, algumas perspectivas futuras estão relatadas abaixo:

- Desenvolver novas funcionalidades e melhorias, como interação entre os usuários;
- Desenvolver uma extensão para *Apple Watch*;
- Adotar novas tecnologias, como *SwiftUI* e programação reativa;

- Aumentar a cobertura de testes;
- Suporte a modo escuro e outros idiomas.

Referências

DUARTE, K.; FALBO, R. Uma ontologia de qualidade de software. 01 2000. Citado na página 16.

HOBSBAWN, E. J. *A Era das Revoluções - 1789 - 1848*. [S.l.]: Paz E Terra, 2012. ? Citado na página 14.

LECHETA, R. R. *Desenvolvimento para iPhone e iPad*. [S.l.]: Novatec, 2017. ? Citado na página 23.

MARTIN, R. C. *Clean Code*. [S.l.]: Prentice Hall, 2009. ? Citado 3 vezes nas páginas 14, 46 e 62.

MARTIN, R. C. *Clean Architecture*. [S.l.]: Prentice Hall, 2018. ? Citado 3 vezes nas páginas 14, 16 e 46.

PEREIRA, C. R. *Construindo APIs REST com Node.js*. [S.l.]: Casa do Código, 2016. ? Citado na página 56.

ROCHA, A. da; MALDONADO, J.; WEBER, K. *Qualidade de software: teoria e prática*. [S.l.]: Prentice Hall, 2001. ISBN 9788587918543. Citado na página 15.

SUTHERLAND, F. S. e J. *Scrum - A Arte de Fazer o Trabalho na Metade do Tempo*. [S.l.]: Sextante, 2019. ? Citado na página 32.