UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE E INOVAÇÃO

MATHIAS GHENO AZZOLINI

Derivação de um estilo arquitetural para o front-end de sistemas baseados em React.js com base em projetos Open Source

Monografia de Conclusão de Curso apresentada como requisito parcial para a obtenção do grau de Especialista em Engenharia de Software e Inovação

Orientador: Prof. Dra. Ingrid Nunes

Porto Alegre
2021

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Azzolini, Mathias Gheno

Derivação de um estilo arquitetural para o front-end de sistemas baseados em React.js com base em projetos Open Source / Mathias Gheno Azzolini. – Porto Alegre: PPGC da UFRGS, 2021.

48 f.: il.

Monografia (especialização) – Universidade Federal do Rio Grande do Sul. Curso de Especialização em Engenharia de Software e Inovação, Porto Alegre, BR–RS, 2021. Orientador: Ingrid Nunes.

1. Arquitetura. 2. React.js. 3. Front-end. I. Nunes, Ingrid. II. Título.

“Good things come,
if you never stop.”

AGRADECIMENTOS

Agradeço à minha família pelo apoio, meus amigos, companheiros do Rotaract e colegas de trabalho pela compreensão das minhas ausências e minha orientadora pela paciência e pela atenção destinada à mim.

RESUMO

O desenvolvimento de aplicações Web ao longo dos anos se tornou mais complexa e a necessidade de se utilizar frameworks para facilitar a criação e manutenção se tornou predominante. Por consequência, alguns frameworks ganharam espaço no mercado: Angular, Vue.js e React.js. O último deles não possui uma arquitetura definida e em comparação com os demais projetos é o mais popular. O objetivo desse trabalho é analisar a arquitetura de projetos Open Source que são desenvolvidos com o framework React.js e extrair elementos comuns para uma proposta de arquitetura. O resultado obtido mostra que os projetos selecionados seguem regras semelhantes ao que diz respeito a separação de conceitos de camada de visual e da camada de manipulação de estado, como também um comportamento semelhante na modularização de funcionalidades.

Palavras-chave: Arquitetura. React.js. Front-end.

ABSTRACT

Over the years, the development of Web applications was become more complex and the need to use an frameworks more common. As consequence, some frameworks become popular: Angular, Vue.js and React.js. The last one doesn't have a defined architecture and is the most popular of them. The main goal of this project is to analyse Open Source projects that use React.js and extract common elements of the architecture. The conclusion of the project show that all projects have common patterns related with visual and state manipulation segregation and a similar patterns for modularity of functionalities.

Keywords: Architecture, React.js, Front-end.

LISTA DE ABREVIATURAS E SIGLAS

SPA	Single Page Application
SSR	Server Side Rendering
OSS	Open Source Software
GUI	Graphical User Interface
NPM	Node Package Manager
ES	ECMAScript
JSON	JavaScript Object Notation
YAML	YAML Ain’t Markup Language
HTML	HyperText Markup Language
API	Application Programming Interface

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Desde a criação das primeiras páginas dinâmicas para a Web com o lançamento do JavaScript em 1995 o desenvolvimento de sistemas Web ficou cada mais complexo e dinâmico (PATIL, 2019). A medida que novas funcionalidades foram adicionadas nos navegadores para suportar o crescimento dos sistemas Web se tornou comum a adoção de frameworks para o desenvolvimento desses sistemas (INZUNZA et al., 2011). Grandes empresas como Google (GOOGLE..., 2021), Facebook (FACEBOOK..., 2021) e Microsoft (MICROSOFT..., 2021) criaram suas próprias tecnologias Open Source para facilitar o desenvolvimento de aplicação na medida que a complexidade dos sistemas Web aumentou. Entre os frameworks que ganharam popularidade destacam-se três: Vue.js [1], Angular [2] e React.js [3], sendo o último o mais popular. Na Figura 1.1, é possível visualizar uma comparação desses três frameworks em relação do número diário de downloads no NPM por um período de 6 meses.

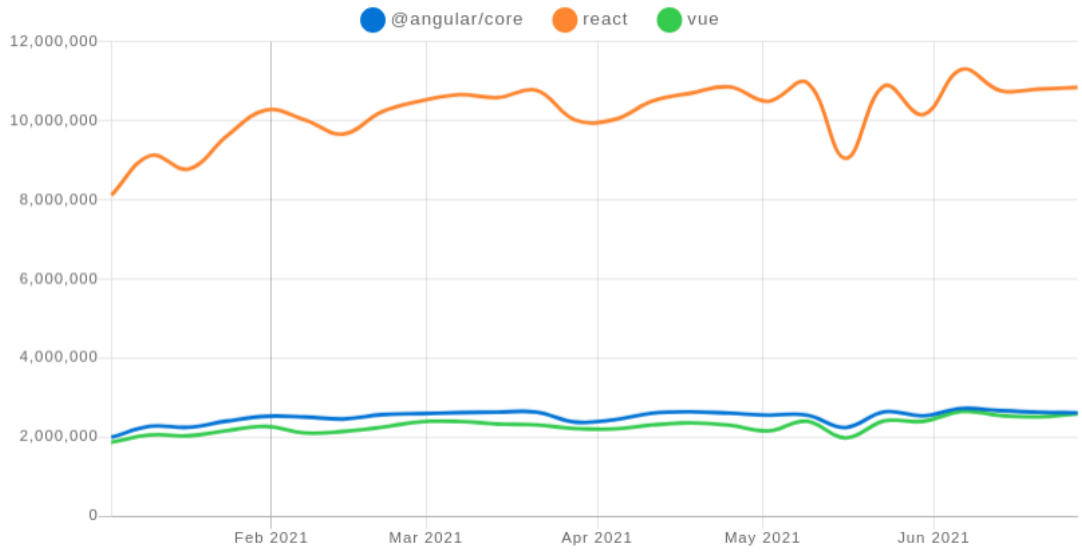
O React.js além de ser o mais utilizado também possui uma flexibilidade alta. Isso porque não existe uma arquitetura de referência para o React.js. Essa falta de arquitetura de referência arquitetural entrega algumas vantagens, como a adoção incremental da tecnologia e a baixa curva de aprendizado. Porém, a falta de uma arquitetura entrega desvantagens, como a falta de padrões de desenvolvimento o que pode prejudicar o manutenção, dificultar o reuso e diminuir a consistência do código (THUNG et al., 2010). Desse modo, é essencial que sejam adotados padrões de desenvolvimento de software e arquiteturas de referência para o desenvolvimento de tais aplicações afim de melhorar a qualidade do projeto.

O objetivo desses trabalho é explorar essas falta de arquitetura para analisar projetos Open Source que utilizam o React.js como fremework base da aplicação. Com essa análise espera-se identificar quais são os padrões de arquitetura utilizados em aplicações React.js para a posterior proposta de uma arquitetura de referência. Os capítulos a seguir deste trabalho estão organizados da forma descrita abaixo:

- O **Capítulo 2** aborda quais são os conceitos teóricos fundamentais para o compreensão desse trabalho.
- O **Capítulo 3** apresenta a metologia utilizada para o desenvolvimento do trabalho.

[1] Vue.js: https://github.com/vuejs/vue
[2] Angular: https://github.com/angular/angular
[3] React.js: https://github.com/facebook/react

Figura 1.1: Comparação de popularidade: Angular, React.js e Vue.js



Fonte: (@ANGULAR/CORE. . . , 2021)

- O **Capítulo 4** apresenta os resultados obtidos através das análise dos projetos selecionados
- O **Capítulo 5** descreve quais são as implicações obtidas a partir das analise arquitetural apresentada no Capítulo 4
- O **Capítulo 6** apresenta as conclusões deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será apresentado os principais conceitos teóricos necessários para compreender o desenvolvimento do trabalho apresentado no Capítulo 5 e Capítulo 3. Serão apresentados as principais caraterísticas da linguagem de programação JavaScript, da arquitetura Redux e do fremework React.js.

2.1 JavaScript

A linguagem de programação JavaScript foi criada em 1995 com o propósito de ser uma linguagem de script simples, tendo como caso de uso validações de formulário e manipulações de DOM que permitiam uma maior dinamicidade de páginas. A linguagem foi criada em 10 dias e desde 1995 vem sendo aprimorada e recebendo atualizações a cada ano (Severance, 2012). JavaScript é o nome comercial utilizado para se referenciar ao ECMAScript (ou ES), uma linguagem de programação neutra, multiplataforma e proposito geral e livre de patentes. A linguagem é especificada pelo TC39, um grupo de trabalho constituído por desenvolvedores e acadêmicos que discutem as novas funcionalidades da linguagem considerando cinco estágios de amadurecimento (TC39, 2021) (ECMA. . . , 2018). A evolução de linguagem tornou possível a criação de novas tecnologias e novos *frameworks* e ferramentas que são popularmente utilizados para facilitar o processo de desenvolvimento de sistemas Web modernos, como o Node.js [1] e o NPM [2].

O Node.js é uma plataforma que possibilita o uso do JavaScript no back-end desenvolvido com base no V8 [3] da Google e tem como característica principal o assincronismo não bloqueante através do uso de eventos para operações de I/O. O Node.js foi lançado em 2012 pelo seu criador, Ryan Dahl (DAHL, 2012). O Node.js possibilitou o surgimento de outras ferramentas, como o NPM. NPM é o acrônimo para *Node Package Manager*, uma software CLI desenvolvido em JavaScript que possibilita o gerenciamento de packages JavaScript (OJAMAA; DüüNA, 2012). O NPM vem instalado de forma embarcada com o Node.js, porém seu uso não é obrigatório, podendo ser utilizado outros gerenciadores de packages como o Yarn [4], PNPM [5].

[1] Node.js: https://github.com/nodejs/node
[2] NPM: https://github.com/npm/cli
[3] V8: https://github.com/v8/v8
[4] Yarn: https://github.com/yarnpkg/yarn
[5] PNPM: https://github.com/pnpm/pnpm

Com o aumento das funcionalidades que linguagem possibilita, tanto para o front-end quanto para o back-end, se tornou comum adoção de bibliotecas terceiras na concepção de novos sistemas. Segundo o estudo de (Mitropoulos et al., 2019) que analisou 10 mil sites concluiu-se que: i) a constância com que o código front-end muda é grande, ii) bibliotecas terceiras são compartilhadas por sites durante um período longo de tempo, iii) vulnerabilidade em bibliotecas tendem a diminuir com o passar do tempo e iv) código relacionado ao conteúdo do site muda com mais frequência que o código base. Além disso, outro estudo analisou o comportamento de depreciações de 50 bibliotecas e concluiu que a frequência com que as Interfaces de desenvolvimento da Aplicação (ou APIs) se depreciam é baixa (Nascimento et al., 2020).

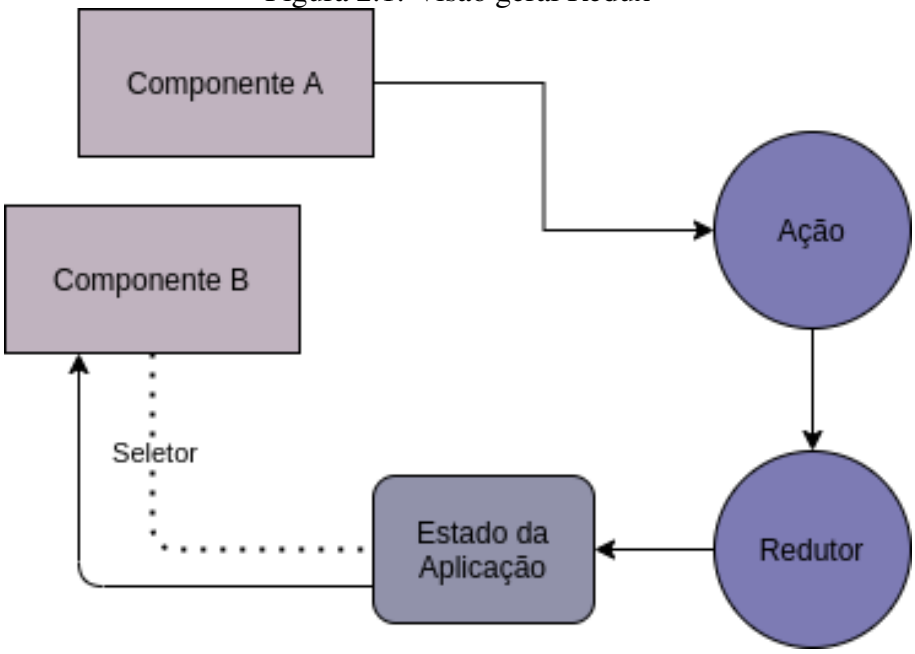
2.2 Redux

Redux é uma arquitetura de manipulação de estado global para gerenciamento de UIs. Foi criada originalmente por Dan Abramov e Andrew Clark em 2015 e foi fortemente inspirada em outra arquitetura de manipulação de estado de autoria do Facebook: o Flux (BACHUK, 2016). O Flux é uma arquitetura de manipulação de estado que possui tres tipos de arquivos: i) Store, ii) Action e iii) Dispatcher (OCCHINO, 2014). Como descrito por um dos criadores da arquitetura em (ABRAMOV, 2016), o Redux é composto essencialmente por três tipos de elementos: i) Store, ii) Actions e iii) Reducers. Na Figura 2.1, é possível visualizar uma visão geral de como o Redux funciona.

A conceito principal do Redux é que o estado de uma aplicação deve ser definido através de um objeto único e global. Além disso, outro conceito fundamental da arquitetura é que todas as mutações que ocorrem no estado global devem ser feitas de maneira declarativa. Somando esses dois conceitos fundamentais, o Redux entrega uma das seus principais vantagens: rastrear todas as mutações que ocorrem no estado global da aplicação e ter uma visualização temporal do comportamento do sistema — o que possibilita ao desenvolvedor e ao usuário final a visualização em etapas de todas as mudanças que ocorreram.

Para representar as alterações que serão feitas no estado da aplicação é necessário a definição de uma "Action", ou uma Ação. Uma Ação é a única operação dentro do sistema que poderá alterar o estado da aplicação, sendo bloqueada qualquer tipo de alteração do estado através de uma outra operação que não seja uma Ação. A estrutura de uma Ação é um objeto JavaScript que deve conter obrigatoriamente uma propriedade chamada `type`

Figura 2.1: Visão geral Redux



que deve ser diferente de `undefined`, sendo as demais propriedade livres. Na Figura 2.2, é possível ver o exemplo de dois tipos de Ações: em `action1` o envio de `nome` e `idade` via demais propriedade do objeto da Ação e em `action2` o envio dos mesmos parâmetros através de uma propriedade do objeto chamada `payload`.

O Redux utiliza conceitos da programação funcionais como a Imutabilidade e Função Pura. Função Pura tem como característica a representação de uma função que utiliza apenas os valores que são passados via propriedade para retorna um novo valor, sem alterar o valor da propriedades ou de alguma variável de escopo superior. Redux utiliza dessa conceito para representar o estado da aplicação. Na Figura 2.3, pode-se notar a declaração de uma Função Pura representada por `fnPura` e uma impura representada por `fnImpura`. Cada nova Ação deve disparar uma função que irá receber o estado atual da aplicação através das propriedade e retornar um novo objeto que representa o novo estado da aplicação. Dentro da arquitetura, a operação responsável por isso é o "Reducer", ou Redutor. Um Redutor é normalmente representado por uma função que recebe dois parâmetros: i) o estado da aplicação e ii) a Ação que será realizada. O corpo do Redutor é comumente representando pela condicional `switch`, porém pode ser representado por condicionais `if`. O conceito de Imutabilidade é caracterizado pela impossibilidade de um valor ser alterado após ele ser criado, sendo possível apenas gerar um novo valor a partir de outro valor.

No Exemplo 2.4 é possível visualizar uma função que representa um Redutor e gerencia uma Ação de adição de usuário no estado da aplicação. Para tornar possível a

Figura 2.2: Exemplo Básico de uma Ação

```
const action1 = (nome, idade) => ({
    type: 'ADICIONAR_USUARIO',
    usuario: {
        nome,
        idade,
    }
})

const action2 = (nome, idade) => ({
    type: 'ADICIONAR_USUARIO',
    payload: {
        nome,
        idade,
    }
})
```

mutação do objeto é necessária a utilização do *Spread Operator*, que tem como responsabilidade a criação de uma nova referência tanto para um array ou objeto. Nota-se que no final é retornado um novo objeto, ou seja, um novo estado para a aplicação — tornando o Redutor uma Função Pura.

O Redutor possui uma assinatura similar ao do método `reduce` do JavaScript, ou seja, recebe uma função acumuladora e um valor inicial. Dentro da arquitetura do Redux o valor inicial é a representação do estado global da aplicação e a função acumuladora é o Redutor, quem define o próximo estado da aplicação a partir de uma Ação. O Redutor é o grande diferencial do Redux quanto comparado com a arquitetura Flux. Como explicado em (ABRAMOV, 2015), o nome Redux vem da soma de Redutor e do Flux, "Reducer"+ "Flux"= "Redux".

Para finalizar o fluxo falta a parte responsável pela apresentação. O estado da aplicação, a Ação e o Redutor precisam trabalhar com uma outra camada: a *view*, ou de Visualização. A camada de Visualização é que irá disparar uma Ação e ser reativa a alterações do estado. Uma abordagem comum é utilizar o *Observer Pattern* para gerenciar isso. Na Figura 2.5, é possível visualizar um exemplo desse padrão.

No exemplo os valores de `redutor` e `action` foram omitidos por serem simulares aos exemplos já apresentados anteriormente. É possível notar a criação de um

Figura 2.3: Exemplo Simples de Função Pura x Função Impura

```
1 function fnPura(nome) {
2     return nome.toLowerCase();
3 }
4
5 function fnImpura(nome) {
6     nome = nome.toLowerCase();
7 }
```

objeto de nome `store` que usa o método `subscribe` para possibilitar a execuções de lógicas a partir da modificação do estado global da aplicação. Também é possível reverter isso executando a função retornada por esse método. Além disso, existem os métodos `getState` e `dispatch`; sendo o primeiro responsável por retornar o estado atual da aplicação e o segundo responsável por executar uma Ação, executar o Redutor e notificar quem estiver observando alterações. Normalmente o termo utilizado em aplicações Redux para identificar qual o estado que está sendo alterado é chamado de *Selector*, ou Seletor. Um Seletor tem como caraterística principal avisar para o observador quando uma propriedade específica do objeto global da aplicação mudou. Na Figura 2.5, podemos visualizar um exemplo de disparo do callback através da observação da mudanças de valores, porém um Seletor iria disparar para uma propriedade específica, como por exemplo apenas para `state.usuarios`.

2.3 React.js

O React.js é um biblioteca JavaScript para desenvolvimento de IUs através da utilização de componentes. O React.js criado por Jordan Walke, então funcionário do Facebook, e foi publicado como ferramenta Open Source em 2013, na JSConf US (OCCHINO; WALKE, 2013). Além de ser compatível com aplicativos Web a biblioteca também pode ser utilizada para o desenvolvimento de aplicativos mobile e no SSR. Em 2015 o time de engenheiros do Facebook tornou Open Source o React Native [6], a principal tecnologia da biblioteca para o desenvolvimento de aplicativos mobile de forma nativa. O React.js já

[6]React Native: https://github.com/facebook/react-native

Figura 2.4: Exemplo Simples de um Redutor

```
const estadoInicial = {
    usuarios: [],
}

function redutor(state = estadoInicial, action) {
    switch (action.type) {
        case 'ADICIONAR_USUARIO':
            const { usuarios } = state;
            const usuarios = [...usuarios, action.payload];
            return {
                ...state,
                usuarios,
            }
    }
}
```

soma mais de 163 mil estrelas, possui cerca de 5.5 milhões de projetos Open Source dependentes do React.js (REACT, 2020) e mais de 9.2 milhões de downloads semanais na NPM (REACT. . . , 2020). Na próxima seção será apresentado as principais características do React.js e como essas características funcionam e como são utilizadas.

2.3.1 Principais Características

Um dos principais características da React.js é o seu uso integrado com o JSX, uma extensão da linguagem que permite a adição de declarações similar ao do XML. Na Figura 2.6, é possível visualizar um exemplo de como o JSX é declarado. Apesar de ser largamente utilizado, o uso do JSX não é obrigatório.

O gerenciamento de estado do React pode ser obtido nativamente através do uso de classes ou de funções. No uso de classes é possível mudar o estado interno de um *component* através do uso do método `setState`. Para mudanças de estado utilizando funções é feito o uso de *hooks* como o `useState`, `useReducer`. Na Figura 2.7, é possível ver um exemplo com classes e na Figura 2.8 é possível ver o mesmo resultado utilizando *hooks*.

Outra característica muito popular do React é a adoção do CSS-in-JS, ou seja a declaração de estilos através do JavaScript. Isso é possível através do uso da propriedade

Figura 2.5: Exemplo Simples de Observer

```
const criarStore = (reducer) => {
    let aux_id = 0;
    let state;
    let listeners = [];
    const getState = () => state;

    const dispatch = (action) => {
        state = reducer(state, action);
        listeners.forEach(listener => listener());
    };

    const subscribe = (listener) => {
        listeners.push(listener);

        const unsubFn = (i) => () => {
            listeners = listeners.filter((_, index) => index !== i);
        }

        const callback = unsubFn(aux_id);

        aux_id = aux_id + 1;

        return callback;
   };
   return { getState, dispatch, subscribe };
};

const store = criarStore(redutor);

const unsubs1 = store.subscribe(() => {
    console.log('estado atualizado:', store.getState())
})
```

`style` que aceita um objeto JavaScript. Na Figura 2.9, é possível visualizar um exemplo do uso do CSS-in-JS de forma nativa para o React. O CSS-in-JS também é adotado através do uso de bibliotecas como o Styled-Components [7] e o Emotion [8].

O React, diferente de outras bibliotecas front-end, adota o fluxo direcional de dados através das *props* dos *components*. Ou seja, numa hierarquia de componentes, componente neto não acessa diretamente os dados do componente pai. No exemplo da Figura 2.10 o Componente Neto A não consegue acessar diretamente dados passados do Componente Pai para o Componente Filho, sendo necessário o Componente Filho passar esses dados de forma explícita para o Componente Neto. Por conta dessa limitação, o React disponibiliza uma funcionalidade chamada *Context* que possibilita o uso de estado compartilhado entre vários componentes independente da hierarquia. Na Figura 2.11, é possível visualizar um exemplo de como o Context, exemplificado como Contexto, muda o comportamento das hierarquias.

[7] Styled Components: https://github.com/styled-components/styled-components
[8] Emotion: https://github.com/emotion-js/emotion

Figura 2.6: Exemplo de uso do JSX

```jsx
const Exemplo = () => (
  <div>
    <h1>Exemplo</h1>
    <p>JS com sintaxe similar ao HTML</p>
  </div>
)
```

2.3.2 Considerações Finais

Dados os fundamentos teóricos apresentados neste capítulo, é possível observar quais são os elementos características do React.js e como que ele pode ser utilizada no desenvolvimento de aplicações Web. Além disso, foi possível observar a evolução da linguagem JavaScript e como isso afetou o processo de desenvolvimento de aplicações Web.

O próximo capítulo irá apresentar quais são os projetos selecionados para a análise arquitetural. Será feita uma breve apresentação das ferramentas e um demonstrativo de visual. No próximo capítulo será demonstrado qual o procedimento que esse trabalho irá adotar para chegar ao resultado esperado.

Figura 2.7: Exemplo mudança de estado com classe

```
class ExempoClasse extends React.Component {
  handleContador() {
    const novoValor = this.state.contador + 1;
    this.setState({ contador })
  }

  render() {
    return (
      <div>
        <p>Contador: {this.state.contador}</p>
        <button
          onClick={this.handleContador.bind(this)}
        >
          Incrementar
        </button>
      </div>
    );
  }
}
```

Figura 2.8: Exemplo mudança de estado com hooks

```
class ExempoClasse extends React.Component {
  handleContador() {
    const novoValor = this.state.contador + 1;
    this.setState({ contador })
  }

  render() {
    return (
      <div>
        <p>Contador: {this.state.contador}</p>
        <button
          onClick={this.handleContador.bind(this)}
        >
          Incrementar
        </button>
      </div>
    );
  }
}
```

Figura 2.9: Exemplo CSS-in-JS com React

```jsx
const ExemploCSS = () => {
  const estiloTexto = {
    textAlign: "center",
    color: '#1e4c87'
  };
  return (
    <div>
      <h1>Exemplo</h1>
      <p style={estiloTexto}>CSS com Objetos JS</p>
    </div>
  )
}
```

Figura 2.10: Exemplo Hierarquia de Componentes

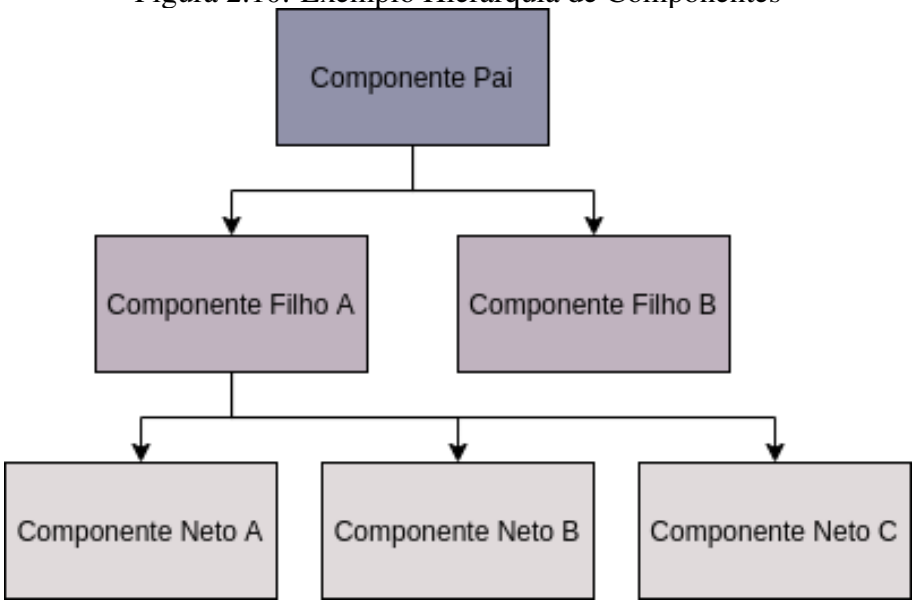
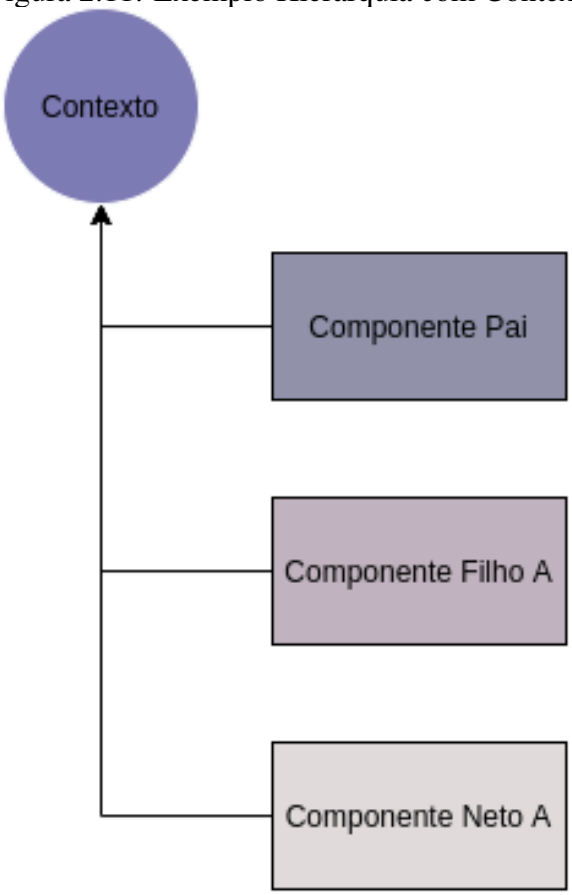


Figura 2.11: Exemplo Hierarquia com Context



3 DETALHAMENTO DO ESTUDO

Nesse capítulo é apresentado o problema que esse trabalho se propõe a resolver, quais são os métodos utilizados para criação da solução final e quais são os projetos selecionados que servem de base para a elaboração do trabalho.

3.1 Questão de Pesquisa

Esse trabalho se propõe a identificar quais são as semelhanças arquiteturais que projetos desenvolvidos em React.js possuem para que seja possível propor uma arquitetura que faça uso dos padrões identificados.

3.2 Procedimento

Para que seja possível identificar uma arquitetura de referência para projetos desenvolvidos em React.js é proposta a análise e seleção de projetos que são desenvolvidos majoritariamente por essa tecnologia. O conjunto total de todos os projetos considerados para a seleção foi obtido através do uso de um *script* que possibilitou selecionar os 100 projetos com maior quantidade de estrelas no GitHub assim como a pesquisa de ferramentas popularmente conhecidas pela comunidade de desenvolvimento. De todos esses projetos obtidos foram selecionados três. O critério de seleção utilizado foi a quantidade de estrelas do projeto pois esse valor indica que o projeto possui uma grande popularidade como também um grande adesão da comunidade de desenvolvimento.

Posteriormente a seleção e divulgação de cada projeto é feita uma coleta de dados para entendimento geral da aplicação. Para fins de analise arquitetural de cada projeto selecionado é considerado o fluxo de dados da aplicação, os tipos de arquivos e suas responsabilidades e por fim a modularidade dos tipos dos arquivos.

Tabela 3.1: Projeto Open Source x Estrelas GitHub

Projeto	Estrelas	Versão
Storybook	59.3k	v6.2.9
Rocket.Chat	29.8k	v3.14.5
Swagger UI	19.7k	v3.50.0

3.3 Projetos de Software Alvo

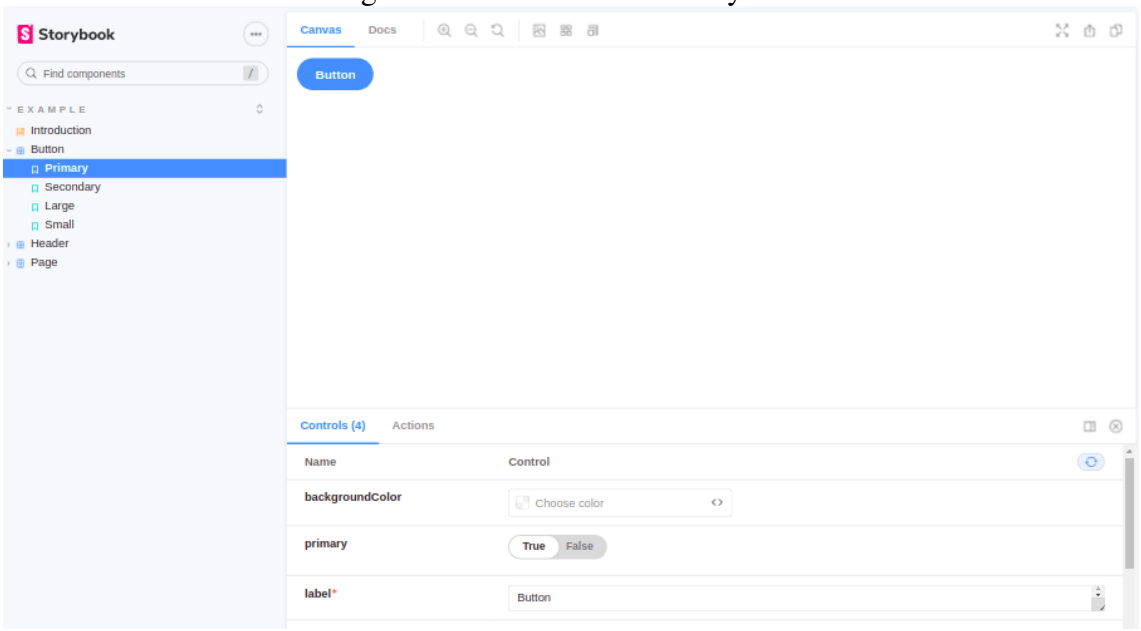
Foram selecionados três projetos Open Source para a análise arquitetural: i) Storybook [1] ii) Rocket.Chat [2] e iii) Swagger UI [3]. A arquitetura desses projetos é analisada nas seção seguinte. O critério escolhido para a escolha desses projetos foi a popularidade do projeto em relação a quantidade de estrelas no GitHub. A Tabela 3.1 contém os valores aproximados da quantidade de estrelas de cada projeto em ordem decrescente e qual a versão de cada projeto e a versão considerada para analise.

O primeiro projeto selecionado é uma ferramenta de definição, desenvolvimento e testes de componentes de UI. A principal funcionalidade da ferramenta é a listagem e renderização de todos os componentes de um projeto numa *viewPort* que pode ser ajustada conforme a necessidade do desenvolvedor e que suporta centenas de extensões que facilitam a documentação, acessibilidade, colaboração e testabilidade (SHILMAN, 2017). Cada componente pode possuir um ou mais estados de renderização. Isso é possível por conta do arquivo de definição de "Stories". O arquivos Stories são definidos através do uso da extensão `.stories` e possui uma retrocompatibilidade com várias frameworks front-end, não sendo sendo restrito ao React.js. O desenvolvimento do plataforma que encorpora o *viewPort* é feita em React.js e será a parte analisada por esse trabalho. Na Figura 3.1, é possível visualizar a tela inicial disponibilizada pela ferramenta.

O segundo projeto selecionado é uma plataforma completa de chat em tempo real e o projeto Open Source mais popular na sua categoria (ENGEL, 2017). A plataforma faz uso de outra tecnologia popularmente utilizada pela comunidade de desenvolvimento JavaScript: o Meteor [4]. O uso do Meteor possibilita a fácil integração do front-end, backen-end e banco de dados numa aplicação única. Um dos principais diferenciais da plataforma é sua integração com diversas plataformas de *deploy* de serviços na nuvem e *containers*. A plataforma é popularmente utilizada de forma interna por empresas (FAL-

[1] Storybook: https://github.com/storybookjs/storybook
[2] Rocket.Chat: https://github.com/RocketChat/Rocket.Chat
[3] Swagger UI: https://github.com/swagger-api/swagger-ui
[4] Meteor: https://github.com/meteor/meteor

Figura 3.1: Tela inicial do Storybook



LAVENA, 2021) e por universidades (ENGEL, 2021).

O terceiro projeto selecionado é uma ferramenta que possibilita a geração automática de documentação para API RESTful. Para que seja possível a geração dessa documentação o projeto faz uso do OpenAPI, um protocolo de documentação de APIs que pode ser escrito em JSON ou em YAML. A ferramenta pode ser utilizada internamente por desenvolvedores, porém pode ser disponibilizado de forma embarcada com as APIs RESTFull. Isso é possível por conta do formato como a qual ferramenta é disponibilizada: HTML e JavaScript que rodam no navegador sem configurações adicionais. No Figura 3.3 é possível visualizar a tela inicial disponibilizada pela ferramenta.

3.4 Considerações Finais

Nesse capítulo foram apresentadas as questões de pesquisa, os procedimentos adotados para o desenvolvimento deste trabalho e quais são os projetos selecionados que serão arquiteturalmente analisados. O próximo capítulo aborda o funcionamento interno de cada aplicação no que diz respeito ao entendimento da arquitetura do projeto.

Figura 3.2: Tela inicial do Rocket.Chat

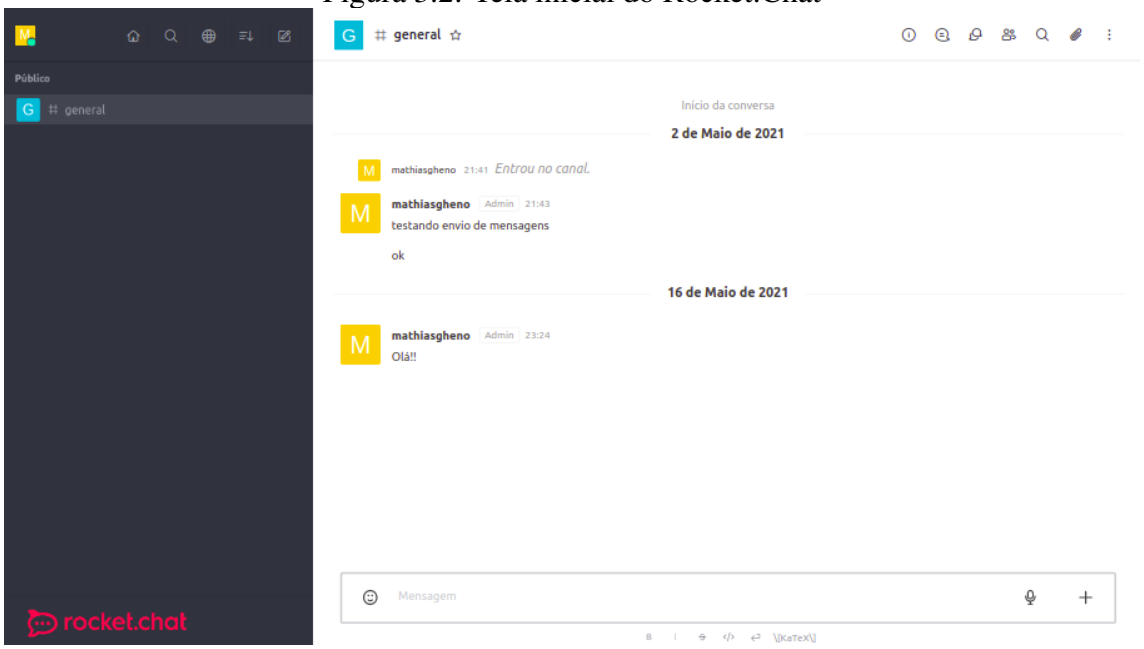
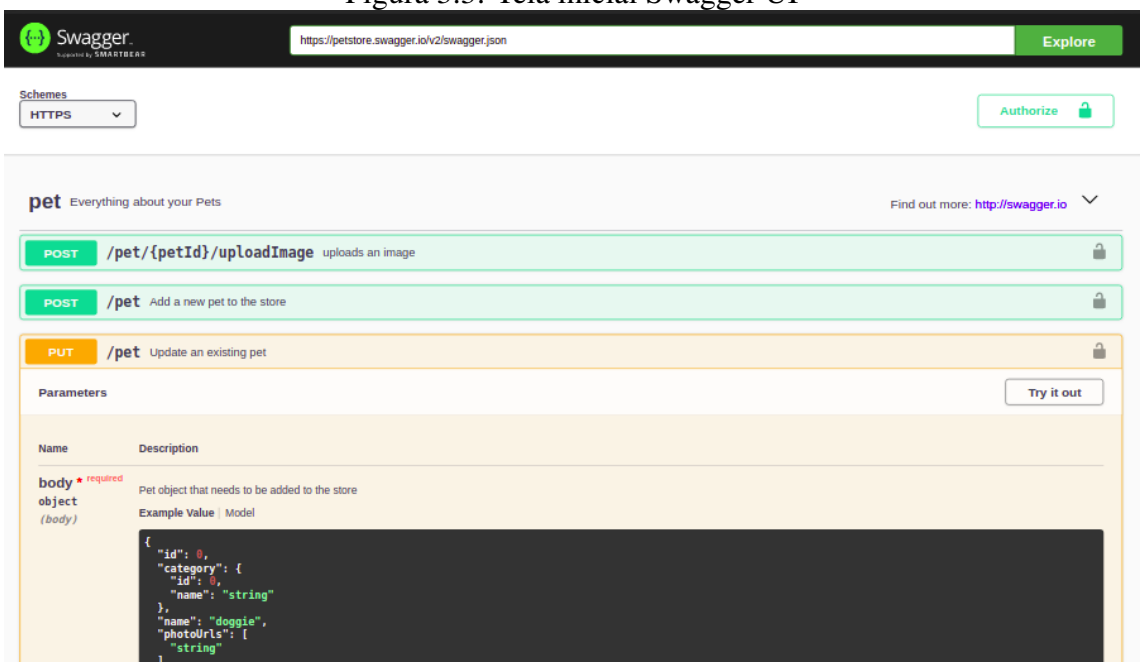


Figura 3.3: Tela inicial Swagger UI



4 RESULTADOS

Nesse capítulo é apresentado o resultado obtido através da análise da arquitetura dos projetos que foram selecionados e apresentados no capítulo anterior.

4.1 Storybook

Nessa seção é apresentado os tipos de arquivos que existem no projeto Storybook. Considerando os tipos de arquivos é feita uma analise das responsabilidades de cada arquivo e suas dependências. Além disso, nessa seção é analisado o comportamento de uma funcionalidade do Storybook para o entendimento do fluxo de dados.

A estrutura do Storybook pode ser dividida em tais tipos de arquivos: i) *components*, ii) *containers*, iii) *stories* e iv) arquivos de configuração e de *mock* de dados. O projeto faz uso de diversos packages que também fazem parte da aplicação. A aplicação é representada pelo package `@storybook/ui`, onde é criado o corpo da ferramenta. Na Tabela 4.1, é possível visualizar uma lista contendo os principais packages que são utilizados pela UI e que são desenvolvidos internamente pelo projeto. Já na Figura 4.1, é possível visualizar quais são os packages que são dependentes da `@storybook/ui`.

O package `@storybook/api` é responsável por definir um conjunto de *hooks* que são utilizados para acesso a estado global da aplicação. Além disso, o package disponibiliza o componente `ManagerProvider` e o `ManagerConsumer`. O primeiro é responsável por criar um *React Context* e providenciar um estado único a partir da API e do estado individual de cada módulo. Cada módulo é definido pelo próprio package e tem como característica a definição de tais propriedades de objeto: i) uma função `init`, ii) um objeto `state` e um objeto `api`. O `ManagerProvider` é utilizado pela UI na renderização do componente *root*. Tento o `ManagerConsumer` quando seu equivalente em *hooks* — o `useStorybookApi`, são utilizados pelos *containers* da aplicação.

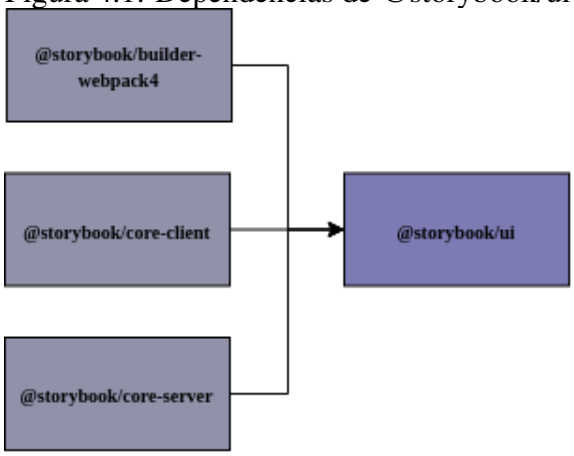
O package `@storybook/router` é quem define os principais *components* que gerenciam as rotas da aplicação. Ele tem como como objetivo ser um wrapper global da dependência externa `@reach/router` e passar o `storyId` vai *props* — que é extraído através da URL da aplicação. Além disso, define um conjunto de utilitários para o gerenciamento de URLs.

O package `@storybook/addons` é responsável pela definição da API pública que possibilita a escrita de extensões para a aplicação. O package disponibiliza um objeto

Tabela 4.1: Dependências da UI

Package
@storybook/addons
@storybook/api
@storybook/channels
@storybook/components
@storybook/core-events
@storybook/client-logger
@storybook/router
@storybook/theming

Figura 4.1: Dependências de @storybook/ui

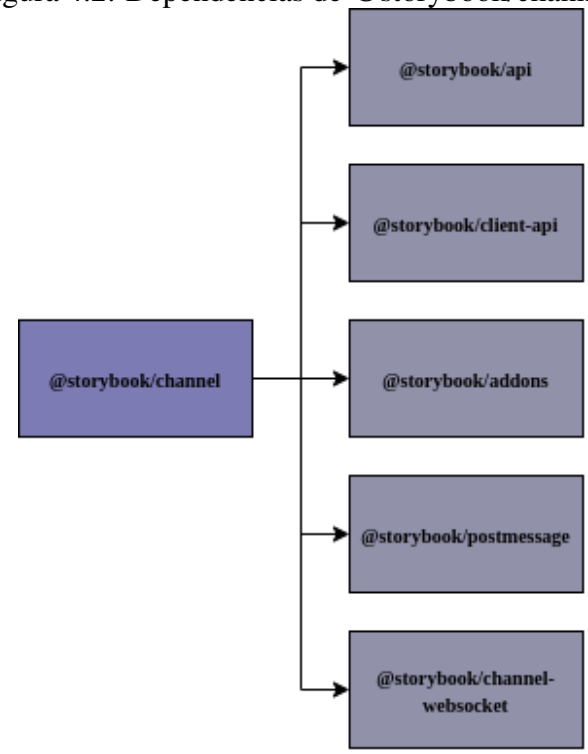


chamado `addons` que possuí três métodos: i) `register` , ii) `add` e iii) `getChannel`. O primeiro método é responsável por fazer o registro da extensão, já o segundo é responsável por adicionar um tipo de elemento na UI do storybook — são permitidos 6 tipos. Por fim o ultimo método é responsável por disponibilizar o acesso a um `EventEmitter` público. Além disso, o package disponibiliza um conjunto de *hooks* para serem utilizados na construção das extensões.

O package `@storybook/channels` disponibiliza um canal de comunicação para envio de eventos entre partes internas do sistemas que, diferentemente do `getChannel` do package `@storybook/addons`, implementa uma versão mais completa e de uso interno da aplicação. O package exporta uma classe `Channel` que é utilizado por diversos packages internos, como extensões de suporte oficial do Storybook — que são definidos dentro do mesmo repositório. Na Figura 4.2, é possível visualizar quais são os packages que são dependentes dele.

O package `@storybook/client-logger` é responsável por exportar um *wrapper* do objeto `console` e duas funções que servem como utilitário: `once` e `pretty`. A primeira função faz o *log* da mensagem apenas uma vez sem adição de estilos, já a se-

Figura 4.2: Dependências de @storybook/channels



gunda função faz o *log* da mensagem utilizando estilos CSS — funcionalidade suportada apenas pelos navegadores.

Os packages `@storybook/theming` e `@storybook/core-events` são mais simples que os demais packages. O primeiro é responsável por exportar configurações que são utilizados para criar o visual da ferramenta, sendo a principal dependência externa o package `emotion` — responsável por gerenciar o CSS-in-JS da aplicação. Já o segundo package é responsável por exportar um conjunto de constantes do tipo *string* que são utilizados pelos demais packages para disparo ou escuta de eventos.

Os *components* são arquivos que representam o visual da aplicação. A maioria deles são componentes sem estado que utilizam o *type* `FunctionComponent`. Os demais estendem `Component` e utilizam manipulação de estado interno através do uso do `setState`. As dependências externas em sua grande maioria são packages internos da própria aplicação, sendo poucas as dependências globais — `global` é o mais utilizado deles. Os packages internos em destaque são: `@storybook/theming`, `@storybook/api` e `@storybook/addons`.

Os *containers* são arquivos utilizados para gerenciar o estado da aplicação de componentes visuais que são base da estrutura visual da aplicação. Todos fazem uso do package `@storybook/api` para acessar o `Consumer` — um componente que faz uso do padrão *Render Props* para passar o estado da aplicação via *callback*. O `Consumer` re-

cebe via *props* uma função responsável por filtrar qual o estado que o componente visual irá receber. Todo container possui uma função de mapeamento do estado, o retorno de um componente visual com o estado mapeado e o uso do `React.memo` ou do `useMemo` para memorização do componente e de suas propriedades.

Os arquivos de stories são arquivos de configuração de renderização de diversos estados visuais de um componente. Um aquivo contendo um `.stories` é utilizado como resultado final para a renderização dos próprios componentes da ferramenta dentro do produto final da ferramenta. Esses arquivos estão presente quase que exclusivamente para os arquivos que representam o visual da aplicação. Cada arquivo possui uma dependência com o React.js e uma dependência com o componente na qual será renderizado os estados visuais.

A funcionalidade que foi escolhida para análise foi a renderização de um story através de canvas da aplicação. Essa funcionalidade é implementa principalmente pelo uso dos packages `@storybook/ui`, `@storybook/api` e `@storybook/router`. O terceiro package é fundamental para a renderização de um novo canvas já que é ela que encapsula o `Preview`, responsável por criar o canvas que irá renderizar o story selecionado pelo usuário. Quem faz o gerenciamento da operação é o `@storybook/api`, que irá fornecer o *hooks* `selectStory` que será responsável por disparar mudanças na store da aplicação assim como executar o `navegate`, responsável por recarregar dados do canvas. Na Figura 4.3, é possível visualizara um diagrama de sequência da funcionalidade.

4.2 Rocket.Chat

Nessa seção é apresentado os tipos de arquivos que existem no projeto Rocket.Chat. Considerando os tipos de arquivos é feita uma analise das responsabilidades de cada arquivo e suas dependências. Além disso, nessa seção é analisado o comportamento de uma funcionalidade do Rocket.Chat para o entendimento do fluxo de dados.

A estrutura do Rocket.Chat pode ser dividida em quatro tipos de arquivos: i) *components*, ii) contexts e providers, iii) hooks, e iv) views. Nos próximos parágrafos é detalhado o uso e responsabilidade desses tipos de arquivos de forma mais detalhada. A UI da aplicação é dependente do Fuselage, outro projeto desenvolvido pela mesma equipe e que possui diversos packages. Na Tabela 4.2, é possível visualizar quais são os principais packages do Fuselage utilizados pela UI.

Figura 4.3: Diagrama Sequência Renderização de Story

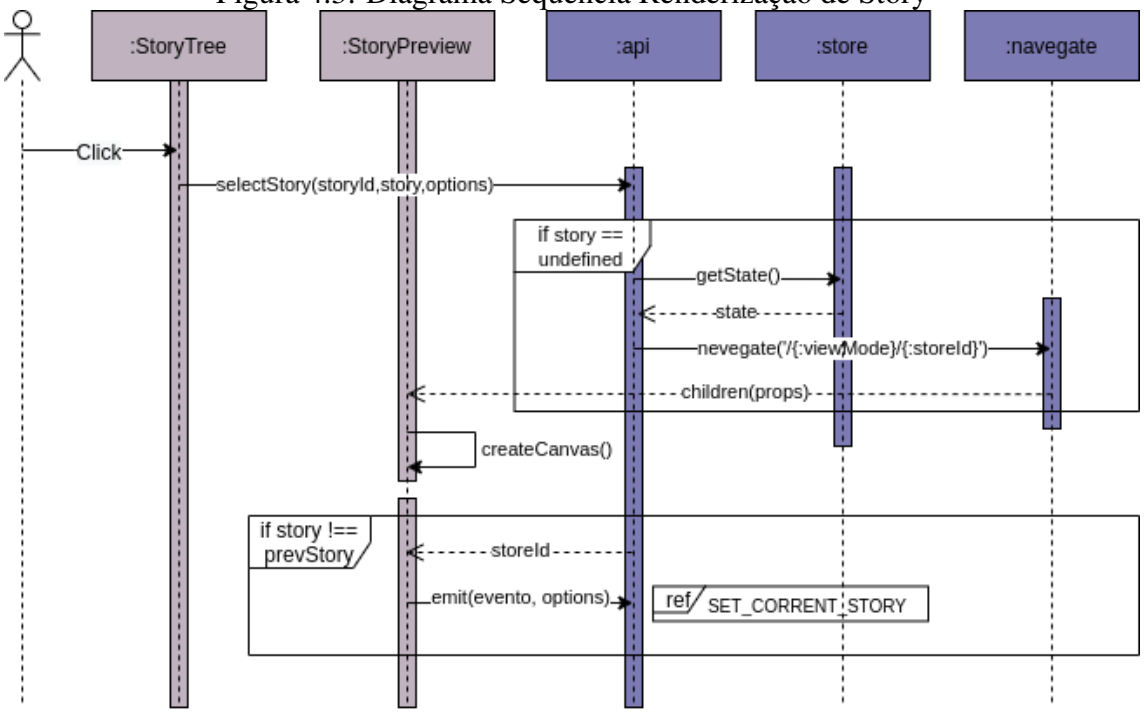


Tabela 4.2: Packages do Fuselage utilizados pela UI

Package
@rocket.chat/css-in-js
@rocket.chat/emitter
@rocket.chat/fuselage
@rocket.chat/fuselage-hooks
@rocket.chat/fuselage-tokens
@rocket.chat/fuselage-ui-kit

O package `@rocket.chat/fuselage` é responsável pela definição de componentes visuais da aplicação. Todos os componentes são definidos utilizando o *type* `FunctionComponent`. Os estilos são definidos utilizando CSS-in-JS através do uso de outro package, o `@rocket.chat/css-in-js` que é acessado via *hooks*. Além do `@rocket.chat/css-in-js` também são utilizados outros dois packages internos: o `@rocket.chat/fuselage-tokens` e o `@rocket.chat/memo`.

O package `@rocket.chat/css-in-js` é responsável por definir um conjunto de utilitários para possibilitar o uso de estilos e também a criação de animações. Além disso, o package é um *wrapper* da dependência externa `stylis`, que tem como objetivo possibilitar a definição de CSS no JavaScript através do uso do *Tag Tamplete String* do ES6. A única dependência interna utilizada pelo package é o `@rocket.chat/memo`.

O package `@rocket.chat/emitter` é reponsável por exportar uma classe `EventEmitter` de forma nativa. Quanto criada a instância do objeto são disponibiliza-

dos três métodos principais: i) `emit`, ii) `on` e iii) `off`. O primeiro método é responsável por emitir um evento, o segundo para definir o *callback* a ser disparado caso um evento ocorra e o terceiro método é responsável por remover o *callback*. O package não faz uso de nenhuma dependência interna ou externa.

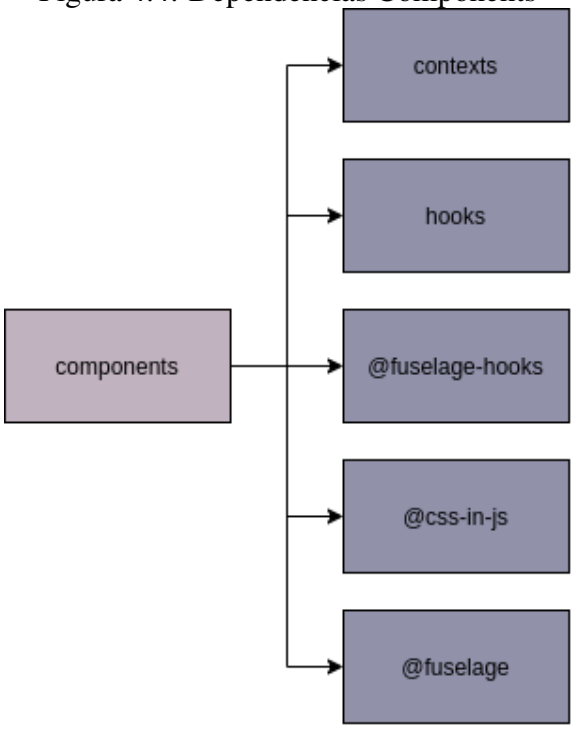
O package `@rocket.chat/fuselage-tokens` é responsável por exportar um conjunto de valores que são utilizados para parametrizar o visual e o comportamento da aplicação de forma global. O package define valores para breakpoints de resolução, cores e tipografia da aplicação. O package não faz uso de nenhuma dependência interna ou externa.

Por fim, o package `@rocket.chat/fuselage-hooks` é responsável por definir um conjunto de *hooks* que possuem funcionalidades variadas, como por exemplo: copiar texto, debounce de alteração de estado, auto-foco de input. O package faz uso de apenas uma dependência interna, o `@rocket.chat/fuselage-tokens`.

Os *components* da aplicação são funções do tipo `FunctionalComponent` e possuem como característica principal a representação visual do sistema para elementos reaproveitáveis. Além de condicionais de renderização, uso de *hooks*, utilitários, esses tipos de arquivos também fazem uso do `Context` para gerenciamento de estado interno do componente. Esses arquivos também são acoplados com outros arquivos do mesmo tipo e com packages internos como o `@rocket.chat/css-in-js`, `@rocket.chat/fuselage` e `@rocket.chat/fuselage-hooks`. Alguns *components* possuem como dependências arquivos do tipo Context e utilitários globais da aplicação, como o React Context `useTranslation` e o utilitário `getUserAvatarURL`. Na Figura 4.4, é possível visualizar o comportamento das dependências.

Os arquivos de *context* são responsáveis por definir dados iniciais para a aplicação de forma global, sendo disponibilizados através do uso do *providers*. Em geral, cada *context* possui um *provider*. Esse tipo de arquivo não faz uso de dependências internas. Um provider é um arquivo que possui responsabilidade de encapsular um context passando métodos de acesso a dados via *props*. Os providers são renderizados no componente *root* da aplicação. As dependências internas do providers são utilitários, models, contexts e módulos externos de consulta e configuração do Meteor — framework utilizado no projeto para integrar front-end e back-end. Alguns providers são utilizados na renderização inicial da aplicação, encapsulando os componentes que gerenciam layout e carregam as páginas da aplicação. Na Figura 4.5, é possível visualizar o comportamento das dependências.

Figura 4.4: Dependências Components

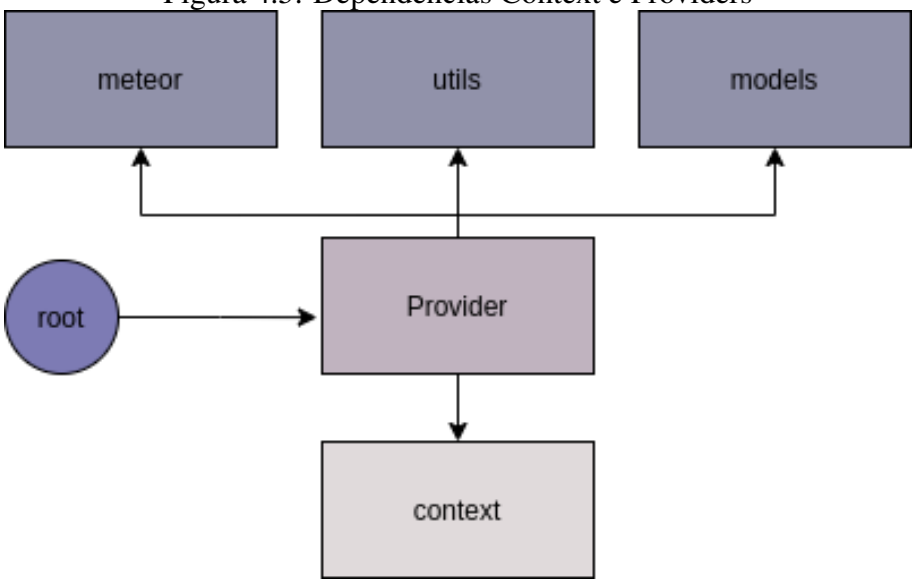


Os arquivos do tipo *hooks* são responsáveis por agregarem utilitários para os componentes em React.js. As funcionalidades variam entre genéricos para o negócio e funcionalidades gerais da aplicação, como requisições e acesso a dados da aplicação. Esses utilitários fazem uso de dependências internas como os arquivos de *Context* e outros *hooks* do package `@rocket.chat/hooks`. Na Figura 4.6, é possível visualizar como é feito o uso desses arquivos em conjunto com os arquivos do tipo view.

As views são arquivos que representam páginas da aplicação. As views, são responsáveis pela renderização de elementos visuais de forma dinâmica e fazem acesso a dependências internas como os hooks, providers, contexts, *components* e aos packages internos como `fuselage`, `fuselage-hooks`, e `fuselage-tokens`.

A funcionalidade que foi escolhida para análise foi o redefinição de senha do usuário através do uso do token. Essa funcionalidade é implementada através do uso de uma view que faz o gerenciamento de quais operações devem ser chamadas. Nessa operação estão envolvidos dois principais arquivos de context, o `serverContext` e o `routerContext`. O primeiro é responsável por fazer a requisição ao backend, já o `serverContext` é responsável por realizar os redirecionamentos necessário caso a operação seja concluída com sucesso. Na Figura 4.7, é possível visualizar um diagrama de dequência que detalha essa funcionalidade.

Figura 4.5: Dependências Context e Providers



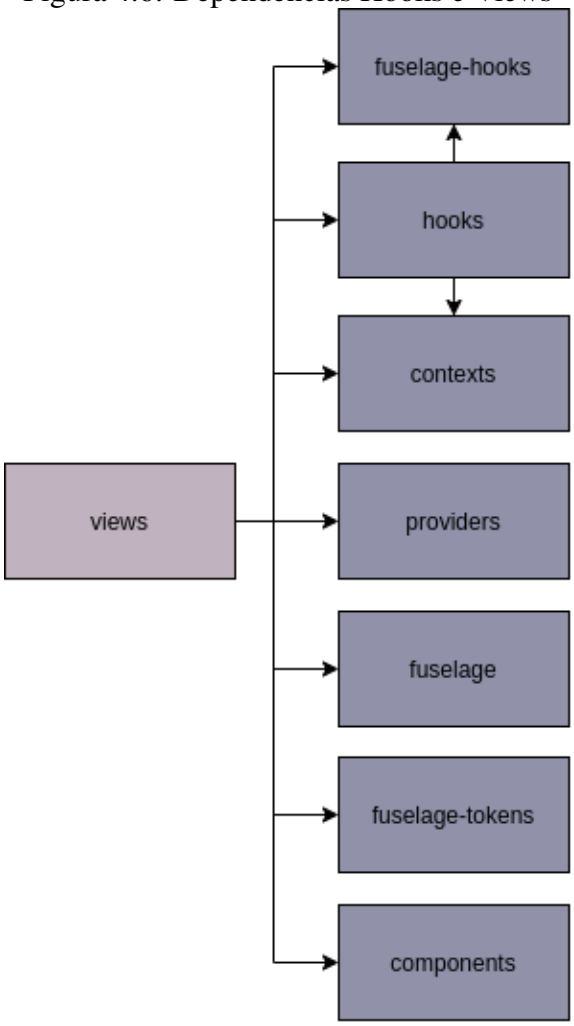
4.3 Swagger UI

Nessa seção é apresentado os tipos de arquivos que existem no projeto Swagger UI. Considerando os tipos de arquivos é feita uma analise das responsabilidades de cada arquivo e suas dependências. Além disso, nessa seção é analisado o comportamento de uma funcionalidade do Swagger UI para o entendimento do fluxo de dados.

A estrutura do Swagger UI pode ser dividida em quatro tipos de arquivos: i) *components*, ii) *containers*, iii) *plugins* e iv) arquivos de configuração e utilitários. Além disso, Swagger UI é uma aplicação SPA que faz o gerenciamento de estado da aplicação através do uso do Redux e da Imutabilidade. Nos próximos parágrafos é detalhado o uso e responsabilidade desses tipos de arquivos de forma mais detalhada. Na Figura 4.8, é possível visualizar a dependência entre esses tipos de arquivos.

Os *components* da aplicação são em maioria classes que estendem `Component`. Esses componentes possuem como característica principal o uso de condicionais de renderização, aplicação de estilos através do uso de classes CSS e validação das propriedades dos componentes através do uso do `defaultProps` e do `propTypes`. Poucos componentes utilizam gerenciamento interno de estado, quando existe a necessidade de se utilizar o gerenciamento de estado é delegada para o estado global da aplicação através do uso das *actions* globais, como, por exemplo do `layoutActions`, `editorActions`, `specActions`. As dependências externas em sua maioria são packages globais da aplicação instalados via NPM. O destaque fica para as dependências: `immutable`, `propTypes` e `react-immutable-proptypes` que são utilizados largamente por

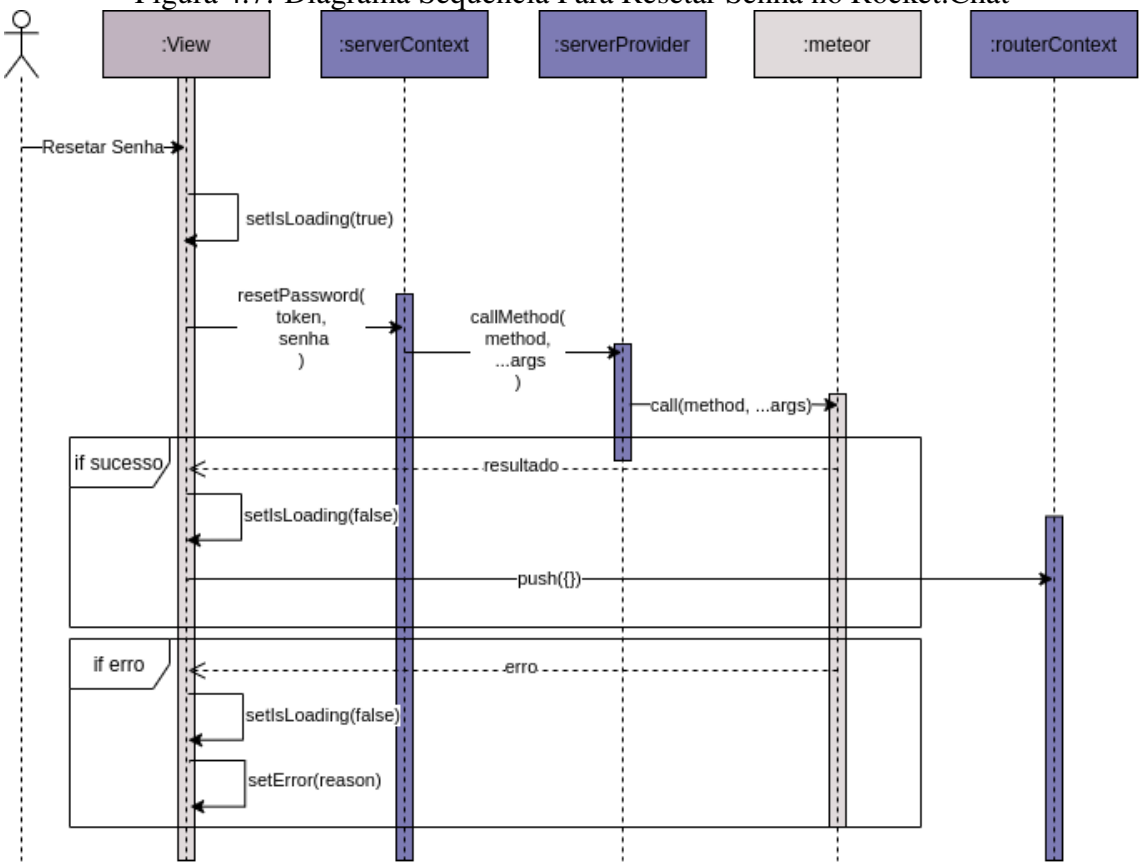
Figura 4.6: Dependências Hooks e Views



esses tipos de arquivos. As principais dependências que não são globais são as dependências relacionadas aos arquivos de utilitários da aplicação.

Os *containers* são tipos de arquivos responsáveis por abstrair o uso de elementos do estado global para os componentes. Os *containers* gerenciam lógicas relacionadas aos *seletores* e as *actions* da aplicação, passando para os componentes renderizados apenas o que eles devem ter acesso. Em sua totalidade todos os *containers* retornam apenas outro componente, sendo o container responsável por *mepear* as propriedades e as passá-las via *props* para o componente retornado. Além disso, os *containers* adicionam condicionais de renderização, protegendo os componentes de serem renderizados caso alguma regra de negócio não seja satisfeita. Todos os *containers* gerenciam componentes que fazem parte do layout base do sistema. Os *containers* possuem dependências externas similares com o que são encontrados nos componentes, sendo as dependências `immutable`, `propTypes` e `react-immutable-proptypes` largamente utilizadas e a dependência com utilitários inexistente.

Figura 4.7: Diagrama Sequência Para Resetar Senha no Rocket.Chat



Os plugins são em sua grande maioria detentores das funcionalidades principais da aplicação. Eles são incorporados dentro do código através do uso do Redux. Cada plugin possui seu conjunto individual de Ações, Redutores e Seletores. Sua inicialização é feita através do arquivo de configuração principal da aplicação: o `Store`. Essa classe é responsável por inicializar e agregar todas as funcionalidades da aplicação, agregando e alterando todos as Ações, os Redutores e os Seletores numa única instância global. As dependências externas variam bastante, porém em sua maioria são dependentes do `immutable` e dos utilitários da aplicação. Os plugins em sua maioria não fazem uso do React.js ou do JSX para manipulação de visual ou lógicas de layout, são responsáveis quase que exclusivamente para a lógica de negócio.

O Swagger UI possui uma funcionalidade que é a possibilidade do usuário disparar uma requisição para a API que está sendo documentada. Essa funcionalidade é implementada através do tipo de arquivo plugin definido pelo nome `spec`. Sendo assim, são implementadas três principais camadas: o container, a ação, o redutor e o estado da aplicação. Sendo a camada de ação a responsável por fazer a requisição assíncrona ao servidor da API. O redutor é o responsável por aplicar a lógica de atualização do estado enquanto o estado da aplicação é quem avisa ao container `executeContainer` que

Figura 4.8: Dependência entre tipos de arquivos Swagger UI

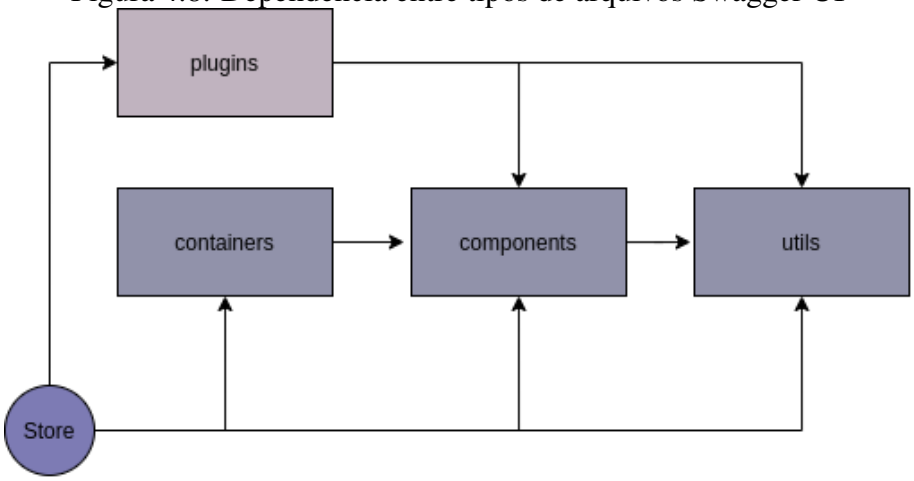
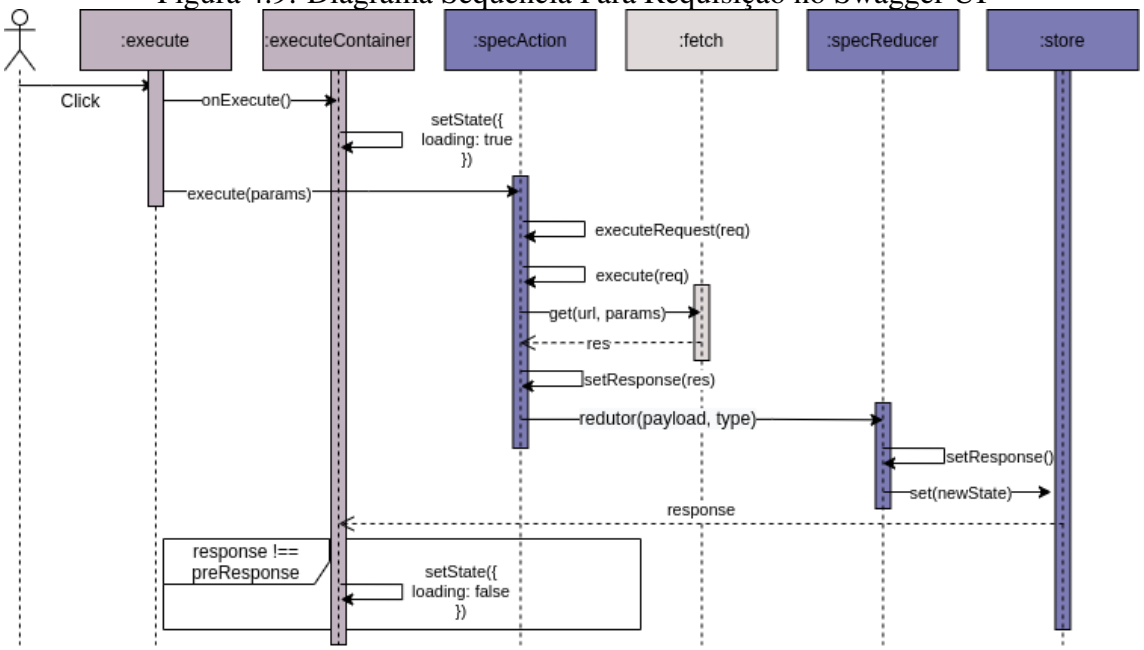


Figura 4.9: Diagrama Sequência Para Requisição no Swagger UI



algo mudou. O container então, renderizada novamente passando os valores para os demais componentes filhos. Na Figura 4.9, é possível visualizar um diagrama de sequência que detalha essa funcionalidade.

5 DISCUSSÃO

Nesse capítulo é discutido quais são as principais características positivas e negativas da arquitetura de cada projeto analisado quanto a sua modularidade. Por fim, como resultado dessa discussão, é apresentada uma proposta de arquitetura.

5.1 Análise da Modularidade Arquitetural

Nessa seção é apresentada a análise arquitetural da modalidade do Swagger UI, Rocket.Chat e do Storybook. A análise arquitetural da modalidade leva em consideração os tipos de arquivos que existem em cada projeto e como eles são modularizados.

5.1.1 Storybook

O Storybook faz uso de uma abordagem semelhante ao do Rocket.chat: modulariza vários elementos da aplicação em diferentes packages que são publicáveis de forma isolada. Porém, nem todos os packages que são utilizados pela UI são reaproveitáveis em, por exemplo, outros projeto. A maioria dos packages são agrupados por tipo de responsabilidade e não por tipo de funcionalidade.

No projeto, operações que são compartilhadas por todos os módulos da aplicação (como a própria aplicação ou os plugins dela) são dependentes de packages que centralizam operações importantes do sistema, como eventos do sistema, roteamento, manipulação de estado ou tematização. Dentre todos os packages o que possui a maior coesão é o `@storybook/api`, pois concentra a manipulação de estado nesse package e disponibiliza o acesso a hooks para todos as partes do sistema que desejam modificar esse estado. Essa abordagem facilita bastante a separação da camada de visualização da camada de lógica da aplicação, pois nenhum arquivo JSX é manipulado. Essa abordagem possibilita que funcionalidades essenciais da aplicação sejam estendida, testadas e migradas de forma isolada e independente da camada visual que está consumindo ao manipulando o estado. Além disso, a abordagem utilizada faz uso de funcionalidades que são embarcadas no React.js, o que faz com que o package tenha um acoplamento bastante reduzido.

Os componentes visuais da aplicação podem ser divididos entre os componentes que são utilizados pela aplicação principal e os componentes que são definidos no

`@storybook/componentes`. Os primeiros são altamente coesão e possuem um acoplamento baixo pois necessitam apenas do `@storybook/theming`. Já os que são definidos dentro do `@storybook/componentes` possuem uma coesão baixa e um acoplamento baixo pois fazem uso de poucas dependências externas ou internas. Agrupar por tipo de arquivo dificulta identificar a qual funcionalidade o componente pertence e aumenta o acoplamento dos dependentes. No caso do Storybook essa prática evita que ocorra uma duplicação de código, pois outros packages fazem uso dos componentes definidos no `@storybook/componentes` justamente por serem genéricos a funcionalidades.

5.1.2 Rocket.Chat

Através do uso do package `@rocket.chat/fuselage*` o projeto consegue fazer o reaproveitamento de muitos componentes da aplicação criando um acoplamento com um conjunto de utilitários, componentes, icones, *hooks* que podem ser reaproveitados pela aplicação inteira. Essa abordagem facilita a migração tecnológica e a modularidade. Diferente da abordagem utilizada pelo Swagger UI o Rocket.chat faz uma segregação por tipo de arquivos (como *hooks* ou *components*) apenas quando eles são genérico, ou seja, esses arquivos podem ser aproveitado fora do projeto original justamente por serem publicáveis para gerenciadores de pacotes como o NPM.

A camada de visual da aplicação é chamada de *view* e representa telas da aplicação. Cada tela possui um conjunto de vários tipos de outros arquivos, o que torna as telas altamente coesas em relação a funcionalidades e acopladas a componentes, hooks e contexts que estão atrelados ao projeto principal de interface. Essa abordagem facilita a identificação de quais são os elementos necessários para que a tela funcione e consequentemente facilita sua modularização e previsibilidade — pois, por exemplo, um container que é definido dentro de um página não será utilizado por outra.

Os componentes de visual da aplicação podem ser divididos entre os componentes que são utilizados pela view e os componentes que são definidos no Fuselage. Ambos possuem um acoplamento baixo com outras camadas do sistema. O componente que é definido dentro da view possui acoplamento com contexts e também com outros componentes que são definidos dentro da própria view ou dentro da biblioteca de componentes. Além disso, esses componentes não possuem estilos próprios, pois fazem uso de outros componentes estruturais que são definidos no Fuselage. Os componentes definidos den-

tro do `@rocket.chat/fuselage` definem seus arquivos de estilo, stories, testes e também types de forma agrupada, o que torna o componente altamente coeso e pouco acoplado por outros arquivos como *helpers* e *utils*.

Dos projetos analisados neste trabalho, o Rocket.Chat e Swagger UI são dependentes de um servidor para funcionar. O Rocket.Chat faz acesso a sua camada de back-end através do uso do `ServerProvider` e do `ServerContext` que são acessados através do React Context na view. O `ServerContext` funciona como um *wrapper* para acesso ao back-end que define todos os endpoints da aplicação em um único local. A camada de view é quem faz a requisição e define qual método irá chamar, mas os métodos disponíveis estão definidos e centralizados dentro do `ServerContext`. Essa abordagem possibilita que hooks como o `useMethod` — utilizado para fazer as requisições, seja acessada em qualquer camada da aplicação como também aumenta o acoplamento de uma view com uma camada definida a nível de aplicação. Essa abordagem prejudica a separação de camadas da aplicação justamente por tornar global o acesso aos métodos e a definição dos métodos disponíveis não resolve problemas relacionados a duplicação de código, o torna o `ServerContext` não coeso.

5.1.3 Swagger UI

Quase todos os tipos de arquivos que existem no projeto estão agrupados com base no tipo do arquivo e não pela sua funcionalidade. Tornando a maioria desses arquivos pouco coesos. Porém, existe um tipo de arquivo que se comporta de forma diferentes: o plugin, que interage diretamente com a Store e permite uma separação da camada de lógica de aplicação da camada visual.

Esses tipos de arquivos são agrupados internamente pela funcionalidade, tornando o plugin altamente coeso e pouco acoplado. Na Figura 4.9, explicada no capítulo anterior, é demonstrado como que a funcionalidade de um plugin interage internamente com arquivos do tipo Action, Reducer, Selector. Isso possibilita que o plugin seja testado, modificado e estendido. Além disso, essa abordagem possibilita que esses plugins possam ser modularizados para packages e reaproveitado por outros projetos ou outros packages da própria aplicação.

Os arquivos do tipo *containers* são arquivos que fazem uso da estado global da aplicação para separar a camada de visual da camada de negócio. Isso possibilita que o componente que representa o visual interaja com dados que foram passados via *prop* pelo

container. Esse estilo *facede* possibilita centralizar operações importantes para UI num único agrupador e ainda possibilita a criação de uma camada de proteção e abstração para o visual o que facilita mudanças no contrato de uma Action ou Selector.

Os arquivos do tipo component possuem a mesma característica: são altamente acoplados com outros tipos de arquivos, como os arquivos de estilos e em alguns casos com outros *components*. Muitos dos estilos não são reaproveitados, sendo usados apenas em um local e declarados em outras locais do projeto. Um exemplo é o component `authoriza-btn`, um component sem estado e que possuí arquivos de estilos declarados fora do próprio componente e que são utilizados apenas uma vez. Essa abordagem dificulta a identificação de que estilo pertence a qual component, aumenta a acoplamento e dificulta a modularização.

5.2 Comparação entre os Projetos

Considerando os elementos discutidos anteriormente, nota-se que todos os projetos possuem abordagens semelhantes como separação da camada de UI e da camada de gerenciamento de estado. No Swagger UI, isso é evidente no uso dos plugins e no uso dos *containers* da aplicação. O Storybook também faz essa separação através do uso do package `@storybook/api`. O Rocket.Chat também faz uso da manipulação do estado da aplicação através do uso de *hooks* que acessam diversos arquivos do tipo *Context*. Porém, diferentemente do Storybook que externaliza isso para um package o Rocket.chat faz a manipulação de estado a nível de aplicação. A manipulação de estado é feita de forma diferente em todos os projetos, porém o uso do React Context é mais recorrente.

Todos os projetos analisados adotam uma estratégia semelhante quanto ao uso de componentes que representam a camada visual. Esses elementos são altamente coesos pois possuem arquivos de estilos, testes, stories e lógica de renderização no Rocket.Chat e no Storybook. A grande diferença é a limitação entre o acesso a dados com hierarquia superior. No caso do Swagger UI, essa limitação é feita através do uso de *containers*, que passam via *props* tudo que o componente visual precisa acessar globalmente. Já o Rocket.Chat e o Storybook fazem o uso do *React Context*. Nota-se também que existem dois tipos de componentes que representam o visual: os internos e externos. Os internos são utilizados apenas uma vez em locais de alta coesão como as páginas da aplicação. Já os externos são modularizados ou segregados afim de serem reaproveitáveis para outros projetos ou para evitar duplicação de código. Essa abordagem é utilizada pelo Rocket.Chat

Tabela 5.1: Comparação entre as abordagens arquiteturais dos três projetos

Abordagem	Swagger UI	Rocket.Chat	Storybook
plugins por funcionalidade	sim	n/a	n/a
providers separando store e visual	n/a	sim	sim
containers separando store e visual	sim	n/a	sim
packages internos publicáveis	n/a	sim	sim
pakages internos publicáveis e genéricos	n/a	sim	não
packages agrupados por tipo de arquivo	n/a	sim	sim
components sem estado	sim	sim	sim
components com estilos próprios	não	sim	sim
views modulares	n/a	sim	n/a
rotas centralizadas	n/a	sim	n/a
manipulação de estado global centralizada	sim	não	sim

no projeto Fuselage e pelo Storybook no package `@storybook/components`.

O Rocket.Chat e o Storybook fazem o isolamento de diversas funcionalidades da aplicação em packages internos que são publicados de forma separada. No caso do Storybook esse package é definido dentro do mesmo repositório e no caso do Rocket.Chat diversos packages utilizados pela UI são definidos num segundo projeto chamado Fuselage. O Swagger UI não adota uma estratégia semelhante referente a packages internos, porém existe algo semelhante: o conceito de plugins — que poderiam ser publicáveis ou internamente modularizados da mesma forma que os demais projetos.

Nem todos os conceitos de um projeto estão presentes em outro projeto, no Rocket.Chat, por exemplo, o conceito de *container* não é utilizado. Outro exemplo é o Swagger UI que não faz uso de packages internos, abordagem utilizado pelo Storybook e pelo Rocket.Chat. Na tabela 5.1, é possível visualizar a listagens das principais abordagens e como elas se relacionam entre os projetos. Conceitos que não estão presentes num projeto são marcados como "n/a".

5.3 Arquitetura Proposta

Com base na identificação de elementos comuns entre os três projetos analisados e com base na discussão feita nesse capítulo foi possível identificar alguns práticas comuns entre os projetos e algumas estratégias arquiteturais que podem ser incorporadas por uma proposta de arquitetura para projetos que tem como tecnologia base o framework React.js. Dentro do contexto da aplicação é possível visualizar 4 grandes camadas: i) container, ii) view, iii) packages publicáveis e iv) packages externos. A descrição da cada

camada é feita no próximos parágrafos e na Figura 5.1 é possível visualizar um resumo da arquitetura proposta.

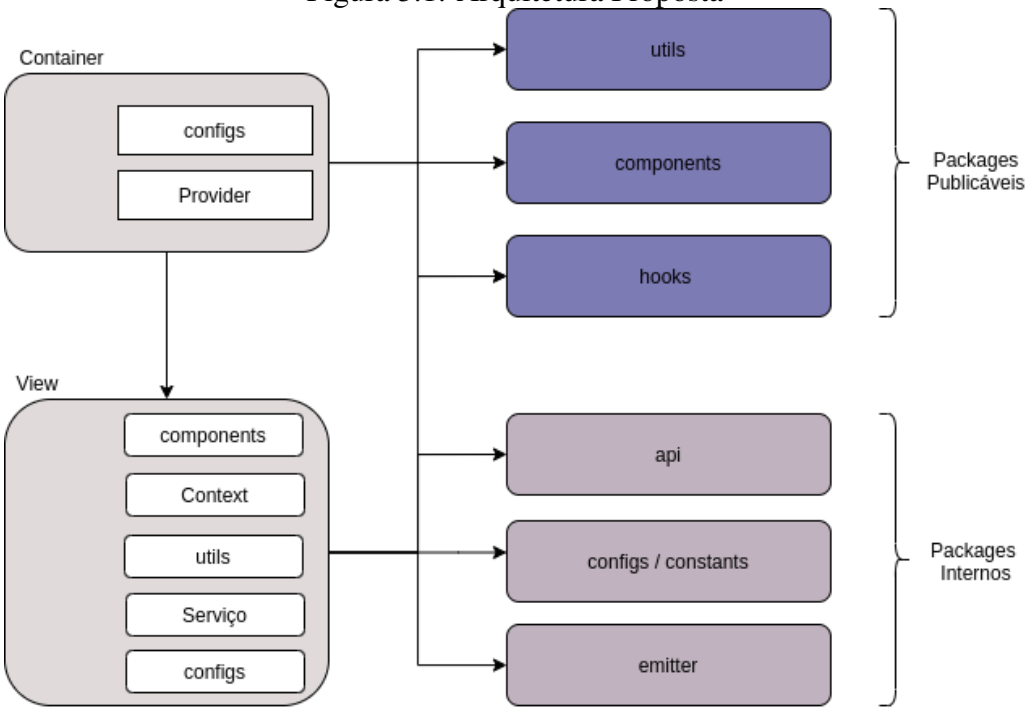
A primeira camada é responsável por fazer o gerenciamento de configuração inicial da aplicação: fornecendo acesso a configurações da aplicação e Providers que irão fornecer React Contexts que podem ser acessados via *hooks* de diversas camadas da aplicação. Nessa camada é possível fazer as configurações que serão a basa do *layout* da plataforma, configuração de rotas e também autenticação.

A camada de view da aplicação é a camada que possui a maior quantidade de tipos de arquivos e responsabilidades. Porém o agrupamento desses arquivos é feito considerando a funcionalidade da aplicação. A view irá conter os arquivos necessários para que ela funciona da forma mais isolada possível, criando acoplamento apenas para evitar duplicação de código, centralização de funcionalidades globais ou para utilitários que são agnósticos a aplicação. Nessa camada é possível gerenciar o que será apresentado para o usuário da aplicação com base nos dados e nas configurações providas via estado local ou global.

A camada de packages interno é o conjunto de tipos de arquivos que podem conter funcionalidades consideradas centrais para a aplicação. Esses packages não devem ser publicáveis pois não podem ser reaproveitados em outras aplicações. Essa camada tem como objeto centralizar operações que permitam a modularidade da aplicação e a orquestração de múltiplos módulos que precisam de um canal comum de comunicação para seu funcionamento. Nessa camada é possível a definição de eventos, *wrappers* de dependências globais, configurações internas da aplicação e gerenciamento de estado global comum entre as views.

A camada de packages publicáveis são elementos que podem ser reaproveitáveis em várias projetos, não apenas nos projetos no qual ele é utilizado. Por conterem arquivos que são agrupados por tipo e que não estão diretamente relacionados com as regras de negocio da aplicação são responsáveis por prover utilitários para as demais camadas da aplicação. Essa camada não difere de uma dependência externa instalada via NPM, sendo o desacoplamento total dessa camada da outras camadas da aplicação totalmente possível e facilitada. Aqui são definidos utilitários, *hooks* que não envolvem funcionalidades específicas da aplicação e componentes reaproveitáveis.

Figura 5.1: Arquitetura Proposta



5.4 Considerações Finais

Nesse capítulos foi discutido as principais características das arquiteturas modulares do Swagger UI, Rocket.Chat e do Storybook. Com base no resultado obtido através da discussão das respectivas arquiteturas foi possível extrair um comparativo entre as aplicações e propor uma arquitetura que pode ser utilizada como base para o desenvolvimento de aplicações com React.js.

6 CONCLUSÃO

Ao longo do trabalho foi demonstrado como as aplicações que são desenvolvidas em React.js apresentam arquitetura diferentes. O principal objetivo do trabalho foi identificar padrões arquiteturais de três projetos Open Source a apresentar uma proposta de arquitetura que pode ser usada como base para o desenvolvimento de aplicações em React.js.

Com base no resultados obtidos nesse trabalho foi possível notar uma separação da manipulação do estado do visual da aplicação. Essa abordagem é utilizada por todos os projetos que foram analisados. A manipulação de estado ocorre de forma diferente para cada aplicação. Porém, os *components* das aplicações são todos visuais, não contendo manipulação de estado ou tendo acesso restrito ao estado da aplicação. Outro resultado obtido foi o uso de packages publicáveis que podem ser segregados em dois tipos: internos e externos. Sendo os packages internos relativos as funcionalidades de negócio das aplicações e os packages externos genéricos as funcionalidades, podendo ser reaproveitados em outas aplicações. A partir da comparação dos três projetos alvo deste trabalho, foi derivada uma arquitetura que satisfaz as principais preocupações de modularidade que estão presentes nestes projetos.

Como trabalho futuro, sugere-se a criação de uma aplicação que faça uso da arquitetura proposta e que possibilite a evolução dessa arquitetura apontando possíveis pontos de melhoria. Além disso, sugere-se que a arquitetura seja testada em aplicações que possuam características funcionais diferentes das analisadas nos projetos selecionados visando estender a adoção da arquitetura.

REFERÊNCIAS

ABRAMOV, D. **Hot Reloading with Time Travel**. 2015. Accessed: 2021-03-28. Available from Internet: <https://www.youtube.com/watch?v=xsSnOQynTHs&ab_channel=ReactEurope>.

ABRAMOV, D. **Fundamentals of Redux Course from Dan Abramov**. 2016. Accessed: 2021-03-27. Available from Internet: <https://egghead.io/lessons/react-redux-the-single-immutable-state-tree>.

@ANGULAR/CORE vs react vs vue. 2021. Accessed: 2021-06-5. Available from Internet: <https://www.npmtrends.com/@angular/core-vs-react-vs-vue>.

BACHUK, A. **Redux · An Introduction**. 2016. Accessed: 2021-03-27. Available from Internet: <https://smashingmagazine.com/2016/06/an-introduction-to-redux/>.

DAHL, R. **Node JS**. 2012. Accessed: 2021-05-23. Available from Internet: <https://www.youtube.com/watch?v=EeYvFl7li9E&ab_channel=JSConfJSConf>.

ECMA International Royalty-Free Patent Policy Extension Option. 2018. Accessed: 2021-05-23. Available from Internet: <https://www.ecma-international.org/policies/by-ipr/royalty-free-patent-policy-extension-option/>.

ENGEL, G. **Como um projeto JS open source se transformou em uma empresa de 60 milhões**. 2017. Accessed: 2021-05-18. Available from Internet: <https://youtu.be/hXG5R15Uc-E>.

ENGEL, G. **How German Universities are improving their communication through Rocket.Chat**. 2021. Accessed: 2021-07-19. Available from Internet: <https://rocket.chat/blog/customer-stories/how-german-universities-are-improving-their-communication-through-rocket-chat/>.

FACEBOOK OSS JavaScript Projects. 2021. Accessed: 2021-06-5. Available from Internet: <https://opensource.fb.com/projects?languages=JavaScript&search=>.

FALLAVENA, L. **Why one of the most important cybersecurity firms in the USA chose Rocket.Chat**. 2021. Accessed: 2021-07-19. Available from Internet: <https://rocket.chat/blog/customer-stories/why-one-of-the-most-important-cybersecurity-firms-in-the-usa-chose-rocket-chat/>.

GOOGLE OSS JavaScript Projects. 2021. Accessed: 2021-06-5. Available from Internet: <https://opensource.google/projects/search?q=javascript>.

INZUNZA, S. et al. Implementing user-oriented interfaces: From user analysis to framework's components. In: **2011 International Conference on Uncertainty Reasoning and Knowledge Engineering**. [S.l.: s.n.], 2011. v. 1, p. 107–110.

MICROSOFT OSS Projects. 2021. Accessed: 2021-06-5. Available from Internet: <https://opensource.microsoft.com/projects>.

Mitropoulos, D. et al. Time present and time past: Analyzing the evolution of javascript code in the wild. In: **2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2019. p. 126–137.

Nascimento, R. et al. Javascript api deprecation in the wild: A first assessment. In: **2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2020. p. 567–571.

OCCHINO, T. **Rethinking Web App Development at Facebook**. 2014. Accessed: 2021-07-31. Available from Internet: <https://youtu.be/nYkdrAPrdcw>.

OCCHINO, T.; WALKE, J. **JS Apps at Facebook**. 2013. Accessed: 2021-02-01. Available from Internet: <https://youtu.be/GW0rj4sNH2w>.

OJAMAA, A.; DüüNA, K. Assessing the security of node.js platform. In: **2012 International Conference for Internet Technology and Secured Transactions**. [S.l.: s.n.], 2012. p. 348–355.

PATIL, H. **Contemporary Front-end Architectures**. 2019. Accessed: 2021-07-05. Available from Internet: <https://blog.webf.zone/contemporary-front-end-architectures-fb5b500b0231>.

REACT. 2020. Accessed: 2021-02-01. Available from Internet: <https://github.com/facebook/react>.

REACT vs @angular/core vs Vue. 2020. Accessed: 2021-02-01. Available from Internet: <https://www.npmtrends.com/react-vs-@angular/core-vs-vue>.

Severance, C. Javascript: Designing a language in 10 days. **Computer**, v. 45, n. 2, p. 7–8, 2012.

SHILMAN, M. **The Storybook Story**. 2017. Accessed: 2021-05-18. Available from Internet: <https://storybook.js.org/blog/the-storybook-story/>.

TC39. 2021. Accessed: 2021-05-23. Available from Internet: <https://www.ecma-international.org/technical-comcmittees/tc39/?tab=general>.

THUNG, P. L. et al. Improving a web application using design patterns: A case study. In: **2010 International Symposium on Information Technology**. [S.l.: s.n.], 2010. v. 1, p. 1–6.