СОВЕРШЕННЫЙ КОД*, ООП*, TYPESCRIPT*, REACTJS*, JAVASCRIPT*

Вам действительно нужен Redux?

Telichkin 12 марта в 19:23 10,8k

Не так давно React позиционировал себя как "V in MVC". После этого коммита маркетинговый текст изменился, но суть осталась той же: React отвечает за отображение, разработчик — за все остальное, то есть, говоря в терминах MVC, за Model и Controller.

Одним из решений для управления Model (состоянием) вашего приложения стал Redux. Его появление мотивировано возросшей сложностью frontend-приложений, с которой не способен справиться MVC.

> Главный Технический Императив Разработки ПО — управление сложностью
>
> — Совершенный код

Redux предлагает управлять сложностью с помощью предсказуемых изменений состояния. Предсказуемость достигается за счет трех фундаментальных принципов:

- состояние всего приложения хранится в одном месте
- единственный способ изменить состояние — отправка Action'ов
- все изменения происходят с помощью чистых функций

Смог ли Redux побороть возросшую сложность и было ли с чем бороться?

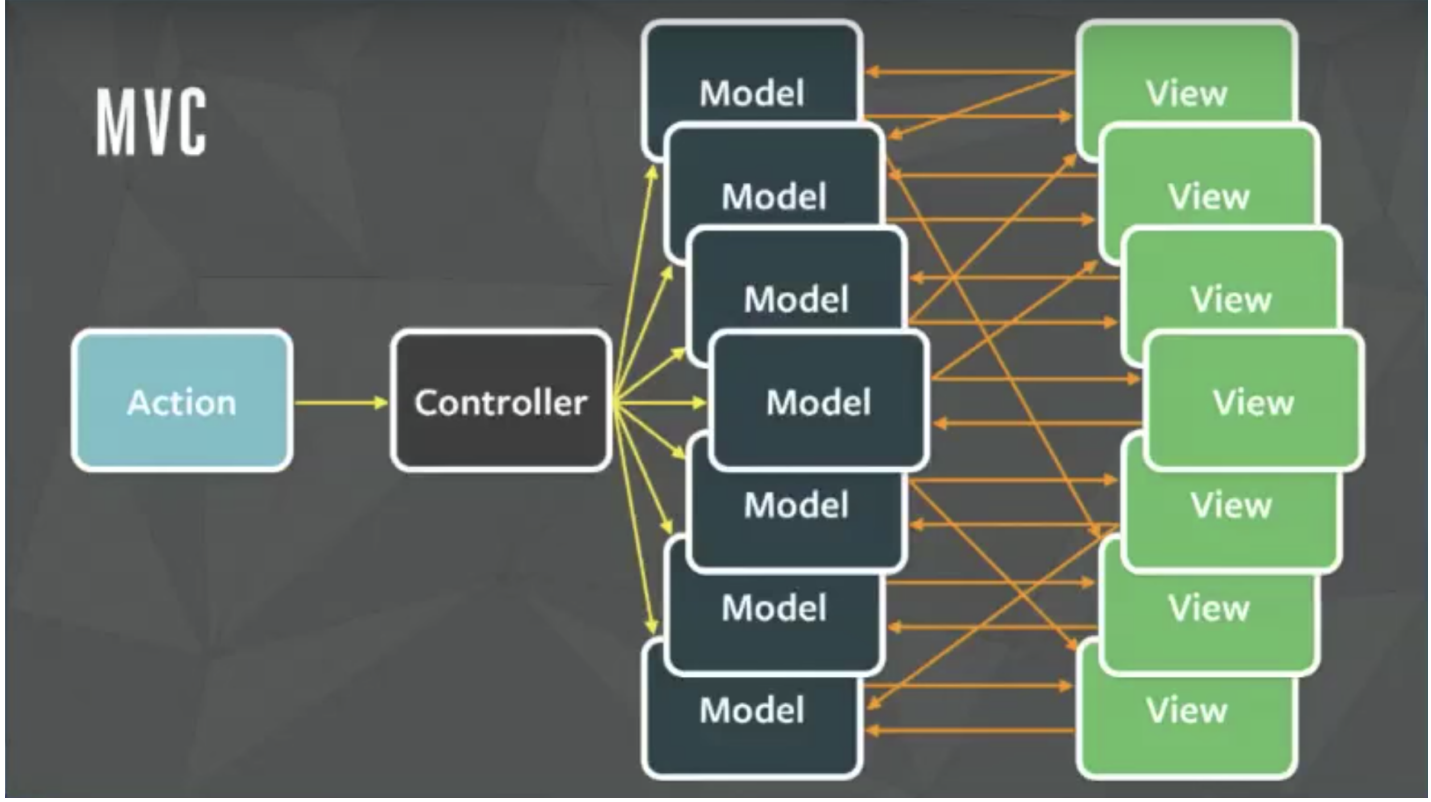
MVC не масштабируется

Redux вдохновлен Flux'ом — решением от Facebook. Причиной создания Flux, как заявляют разработчики Facebook (видео), была проблема масштабируемости архитектурного шаблона MVC.

По описанию Facebook, связи объектов в больших проектах, использующих MVC, в конечном итоге становятся непредсказуемыми:

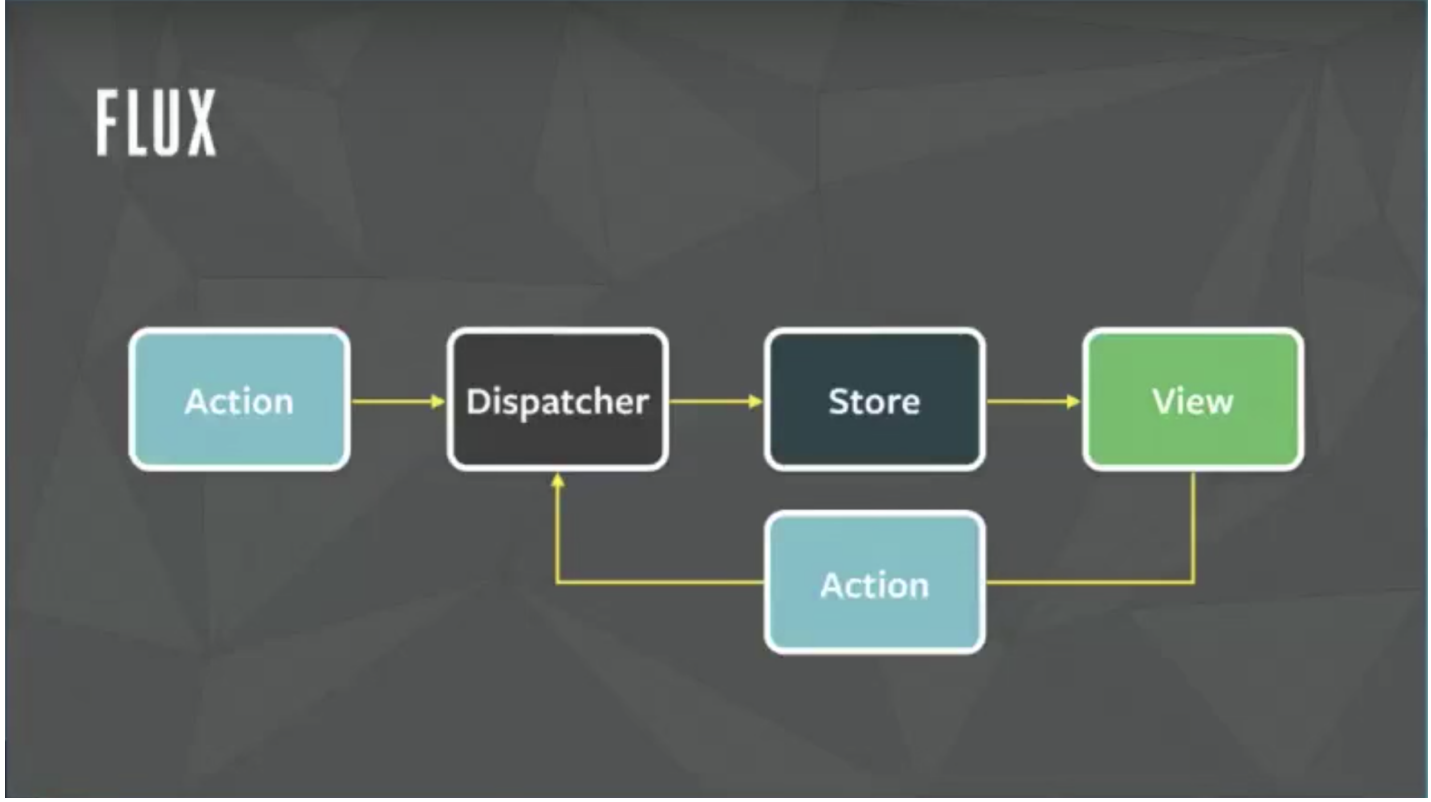
1. modelOne изменяет viewOne
2. viewOne во время своего изменения изменяет modelTwo
3. modelTwo во время своего изменения изменяет modelThree
4. modelThree во время своего изменения изменяет viewTwo и viewFour

О проблеме непредсказуемости изменений в MVC также написано в мотивации Redux'a. Картинка ниже иллюстрирует как видят эту проблему разработчики Facebook'a.

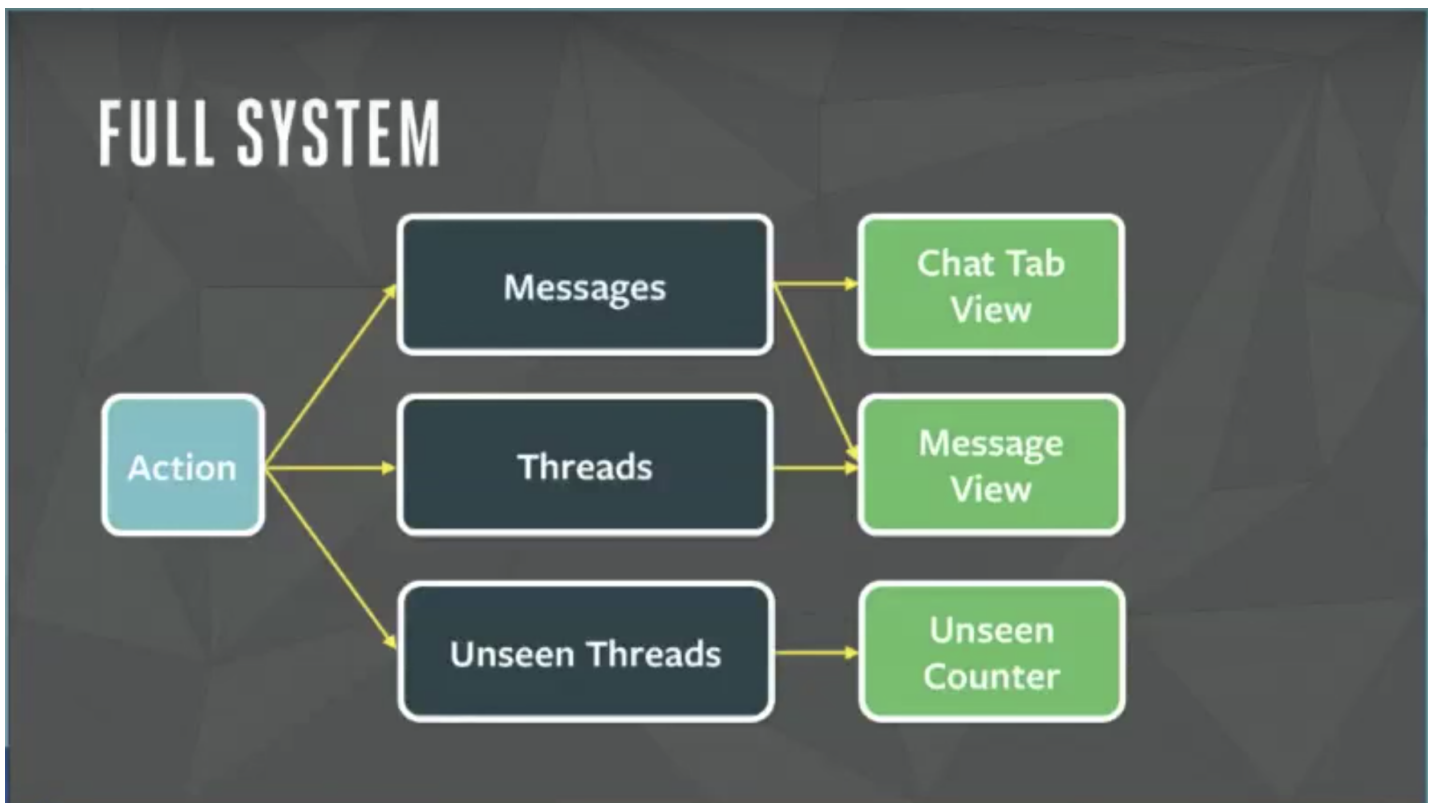


Flux, в отличии от описанного MVC, предлагает понятную и стройную модель:

1. View порождает Action
2. Action попадает в Dispatcher
3. Dispatcher обновляет Store
4. Обновленный Store оповещает View об изменении
5. View перерисовывается



Кроме того, используя Flux, несколько Views могут подписаться на интересующие их Stores и обновляться только тогда, когда в этих Stores что-нибудь изменится. Такой подход уменьшает количество зависимостей и упрощает разработку.



Реализация MVC от Facebook полностью отличается от оригинального MVC, который был широко распространен в Smalltalk-мире. Это отличие и является основной причиной заявления "MVC не масштабируется".

Назад в восьмидесятые

MVC — это основной подход к разработке пользовательских интерфейсов в Smalltalk-80. Как Flux и Redux, MVC создавался для уменьшения сложности ПО и ускорения разработки. Я приведу краткое описание основных принципов MVC-подхода, более детальный обзор можно почитать здесь и здесь.

Ответственности MVC-сущностей:

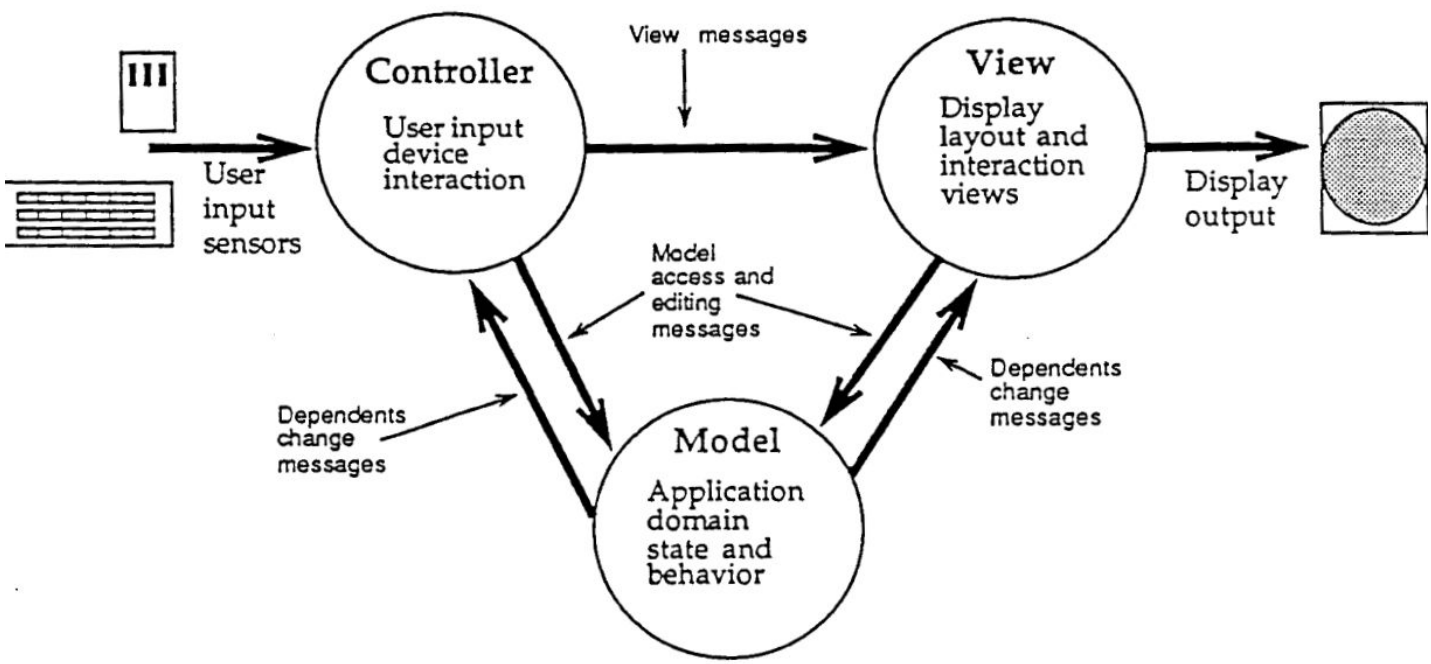
- Model — это центральная сущность, которая моделирует реальный мир и бизнес-логику, предоставляет информацию о своем состоянии, а также изменяет свое состояние по запросу из Controller'а
- View получает информацию о состоянии Model и отображает ее пользователю
- Controller отслеживает движение мыши, нажатие на кнопки мыши и клавиатуры и обрабатывает их, изменяя View или Model

А теперь то, что упустил Facebook, реализуя MVC — связи между этими сущностями:

- View может быть связана **только с одним** Controller'ом

- Controller может быть связан **только с одним** View
- Model **ничего не знает** о View и Controller и **не может их изменять**
- View и Controller подписываются на Model
- Одна пара View и Controller'а может быть подписана **только на одну** Model
- Model может иметь **много подписчиков** и оповещает всех их после изменения своего состояния

Посмотрите на изображение ниже. Стрелки, направленные от Model к Controller'у и View — это не попытки изменить их состояние, а оповещения об изменениях в Model.



Оригинальный MVC совершенно не похож на реализацию Facebook'а, в которой View может изменять множество Model, Model может изменять множество View, а Controller не образует тесную связь один-к-одному с View. Более того, Flux — это MVC, в

котором роль Model играют Dispatcher и Store, а вместо вызова методов происходит отправка Action'ов.

React через призму MVC

Давайте посмотрим на код простого React-компонента:

```
class ExampleButton extends React.Component {
  render() { return (
    <button onClick={() => console.log("clicked!")}>
        Click Me!
    </button>
  ); }
}
```

А теперь еще раз обратимся к описанию Controller'a в оригинальном MVC:

> Controller отслеживает движение мыши, нажатие на кнопки мыши и клавиатуры и обрабатывает их, изменяя View или Model
>
> Controller может быть связан **только с одним** View

Заметили, как Controller проник во View на третьей строке компонента? Вот он:

```
onClick={() => console.log("clicked!")}
```

Это идеальный Controller, который полностью удовлетворяет своему описанию. JavaScript сделал нашу жизнь легче, убрав необходимость самостоятельно отслеживать положение мыши и координаты в которых произошло нажатие. Наши React-

компоненты превратились не просто во View, а в тесно связанные пары View-Controller.

Работая с React, нам остается только реализовать Model. React-компоненты смогут подписываться на Model и получать уведомления при обновлении ее состояния.

Готовим MVC

Для удобства работы с React-компонентами, создадим свой класс BaseView, который будет подписываться на переданную в props Model:

```
// src/Base/BaseView.tsx
import * as React from "react";
import BaseModel from "./BaseModel";

export default class <Model extends BaseModel, Props> extends
React.Component<Props & {model: Model}, {}> {
    protected model: Model;

    constructor(props: any) {
        super(props);
        this.model = props.model
    }

    componentWillMount() { this.model.subscribe(this); }

    componentWillUnmount() { this.model.unsubscribe(this); }
}
```

В этой реализации атрибут state всегда является пустым объектом, потому что мне он показался бесполезным. View может хранить свое состояние непосредственно в атрибутах экземпляра класса и при необходимости вызывать `this.forceUpdate()`, чтобы

перерисовать себя. Возможно, такое решение является не самым лучшим, но его легко изменить, и оно не влияет на суть статьи.

Теперь реализуем класс BaseModel, который предоставляет возможность подписаться на себя, отписаться от себя, а также оповестить всех подписчиков об изменении состояния:

```
// src/Base/BaseModel.ts
export default class {
    protected views: React.Component[] = [];

    subscribe(view: React.Component) {
        this.views.push(view);
        view.forceUpdate();
    }

    unsubscribe(view: React.Component) {
        this.views = this.views.filter((item: React.Component) =>
item !== view);
    }

    protected updateViews() {
        this.views.forEach((view: React.Component) =>
view.forceUpdate())
    }
}
```

Я реализую всем известный TodoMVC с урезанным функционалом, весь код можно посмотреть на Github.

TodoMVC является списком, который содержит в себе задачи. Список может находится в одном из трех состояний: "показать все задачи", "показать только активные задачи", "показать завершенные задачи". Также в список можно добавлять и удалять задачи. Создадим соответствующую модель:

```
// src/TodoList/TodoListModel.ts
import BaseModel from "../Base/BaseModel";
import TodoItemModel from "../TodoItem/TodoItemModel";

export default class extends BaseModel {
    private allItems: TodoItemModel[] = [];
    private mode: string = "all";

    constructor(items: string[]) {
        super();
        items.forEach((text: string) => this.addTodo(text));
    }

    addTodo(text: string) {
        this.allItems.push(new TodoItemModel(this.allItems.length,
text, this));
        this.updateViews();
    }

    removeTodo(todo: TodoItemModel) {
        this.allItems = this.allItems.filter((item: TodoItemModel)
=> item !== todo);
        this.updateViews();
    }

    todoUpdated() { this.updateViews(); }

    showAll() { this.mode = "all"; this.updateViews(); }

    showOnlyActive() { this.mode = "active"; this.updateViews(); }

    showOnlyCompleted() { this.mode = "completed";
this.updateViews(); }

    get shownItems() {
        if (this.mode === "active") { return this.onlyActiveItems; }
        if (this.mode === "completed") { return
this.onlyCompletedItems; }
        return this.allItems;
    }

    get onlyActiveItems() {
        return this.allItems.filter((item: TodoItemModel) =>
item.isActive());
    }

    get onlyCompletedItems() {
        return this.allItems.filter((item: TodoItemModel) =>
```

```
item.isCompleted());
    }
}
```

Задача содержит в себе текст и идентификатор. Она может быть либо активной, либо выполненной, а также может быть удалена из списка. Выразим эти требования в модели:

```
// src/TodoItem/TodoItemModel.ts
import BaseModel from "../Base/BaseModel";
import TodoListModel from "../TodoList/TodoListModel";

export default class extends BaseModel {
    private completed: boolean = false;
    private todoList?: TodoListModel;
    id: number;
    text: string = "";

    constructor(id: number, text: string, todoList?: TodoListModel)
{
        super();
        this.id = id;
        this.text = text;
        this.todoList = todoList;
    }

    switchStatus() {
        this.completed = !this.completed
        this.todoList ? this.todoList.todoUpdated() :
this.updateViews();
    }

    isActive() { return !this.completed; }

    isCompleted() { return this.completed; }

    remove() { this.todoList && this.todoList.removeTodo(this) }
}
```

К получившимся моделям можно добавлять любое количество View, которые будут обновляться сразу после изменений в Model.

Добавим View для создания новой задачи:

```
// src/TodoList/TodoListInputView.tsx
import * as React from "react";
import BaseView from "../Base/BaseView";
import TodoListModel from "./TodoListModel";

export default class extends BaseView<TodoListModel, {}> {
    render() { return (
        <input
            type="text"
            className="new-todo"
            placeholder="What needs to be done?"
            onKeyDown={(e: any) => {
                const enterPressed = e.which === 13;
                if (enterPressed) {
                    this.model.addTodo(e.target.value);
                    e.target.value = "";
                }
            }}
        />
    ); }
}
```

Зайдя в такой View, мы сразу видим, как Controller (props onKeyDown) взаимодействует с Model и View, и какая конкретно Model используется. Нам не нужно отслеживать всю цепочку передачи props'ов от компонента к компоненту, что уменьшает когнитивную нагрузку.

Реализуем еще один View для модели TodoListModel, который будет отображать список задач:

```
// src/TodoList/TodoListView.tsx
import * as React from "react";
import BaseView from "../Base/BaseView";
import TodoListModel from "./TodoListModel";
import TodoItemModel from "../TodoItem/TodoItemModel";
```

```
import TodoItemView from "../TodoItem/TodoItemView";

export default class extends BaseView<TodoListModel, {}> {
    render() { return (
        <ul className="todo-list">
            {this.model.shownItems.map((item: TodoItemModel) =>
<TodoItemView model={item} key={item.id}/>)}
        </ul>
    ); }
}
```

И создадим View для отображения одной задачи, который будет работать с моделью TodoItemModel:

```
// src/TodoItem/TodoItemView.jsx
import * as React from "react";
import BaseView from "../Base/BaseView";
import TodoItemModel from "./TodoItemModel";

export default class extends BaseView<TodoItemModel, {}> {
    render() { return (
        <li className={this.model.isCompleted() ? "completed" : ""}>
            <div className="view">
                <input
                    type="checkbox"
                    className="toggle"
                    checked={this.model.isCompleted()}
                    onChange={() => this.model.switchStatus()}
                />
                <label>{this.model.text}</label>
                <button className="destroy" onClick={() =>
this.model.remove()}/>
            </div>
        </li>
    ); }
}
```

TodoMVC готов. Мы использовали только собственные абстракции, которые заняли меньше 60 строк кода. Мы работали в один момент времени с двумя движущимися частями: Model и View, что

снизило когнитивную нагрузку. Мы также не столкнулись с проблемой отслеживания функций через props'ы, которая быстро превращается в ад. А еще нам не пришлось создавать фэйковые Container-компоненты.

Что не так с Redux?

Меня удивило, что найти истории с негативным опытом использования Redux проблематично, ведь даже автор библиотеки говорит, что Redux подходит не для всех приложений. Если ваше frontend-приложения должно:

- уметь сохранять свое состояние в local storage и стартовать, используя сохраненное состояние
- уметь заполнять свое состояние на сервере и передавать его клиенту внутри HTML
- передавать Action'ы по сети
- поддерживать undo состояния приложения

То можете выбирать Redux в качестве паттерна работы с моделью, в ином случае стоит задуматься об уместности его применения.

Redux слишком сложный, и я говорю не про количество строк кода в репозитории библиотеки, а про те подходы к разработке ПО, которые он проповедует. Redux возводит indirection в абсолют, предлагая начинать разработку приложения с одних лишь Presentation Components и передавать все, включая Action'ы для изменения State, через props. Большое количество indirection'ов в одном месте делает код сложным. А создание переиспользуемых и

настраиваемых компонентов в начале разработки приводит к преждевременному обобщению, которое делает код еще более сложным для понимания и модификации.

Для демонстрации indirection'ов можно посмотреть на такой же TodoMVC, который расположен в официальном репозитории Redux. Какие изменения в State приложения произойдут при вызове callback'а onSave, и в каком случае они произойдут?

При отсутствии желания устраивать расследование самостоятельно, можно заглянуть под спойлер

И это простой случай, потому что callback'и, полученные из props'ов, оборачивались в дополнительную функциональность только 2 раза. В реальном приложении можно столкнуться с ситуацией, когда таких изменений гораздо больше.

При использовании оригинального MVC, понимать, что происходит с моделью приложения гораздо проще. Такое же изменение заметки не содержит ненужных indirection'ов и инкапсулирует всю логику изменения в модели, а не размазывает ее по компонентам.

Создание Flux и Redux было мотивировано немасштабируемостью MVC, но эта проблема исчезает, если применять оригинальный MVC. Redux пытается сделать изменение состояния приложения предсказуемым, но водопад из callback'ов в props'ах не только не способствует этому, но и приближает вас к потере контроля над вашим приложением. Возросшей сложности frontend-приложений, о которой говорят авторы Flux и Redux, не было. Было лишь

неправильное использование подхода к разработке. Facebook сам создал проблему и сам же героически ее решил, объявив на весь мир о "новом" подходе к разработке. Б**о**льшая часть frontend-сообщества последовала за Facebook, ведь мы привыкли доверять авторитетам. Но может настало время остановиться на мгновение, сделать глубокий вдох, отбросить хайп и дать оригинальному MVC еще один шанс?

UPD

Изменил изначальные `view.setState({})` на `view.forceUpdate()`. Спасибо, kahi4.

Похожие публикации

ПЕРЕВОД

AlmazDelDiablo • 17 апреля 2016 в 19:11

Декоратор (Перевод с английского главы «Decorator» из книги «Pro Objective-C Design Patterns for iOS» Carlo Chung)

2

WildCat2013 • 19 февраля 2014 в 03:43

Синглтон (Перевод с английского главы «Singleton» из книги «Pro Objective-C Design Patterns for iOS» Carlo Chung)

10

ИЗ ПЕСОЧНИЦЫ

WildCat2013 • 21 октября 2013 в 20:47

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации 2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству? 8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32