

Как прекратить войну фреймворков

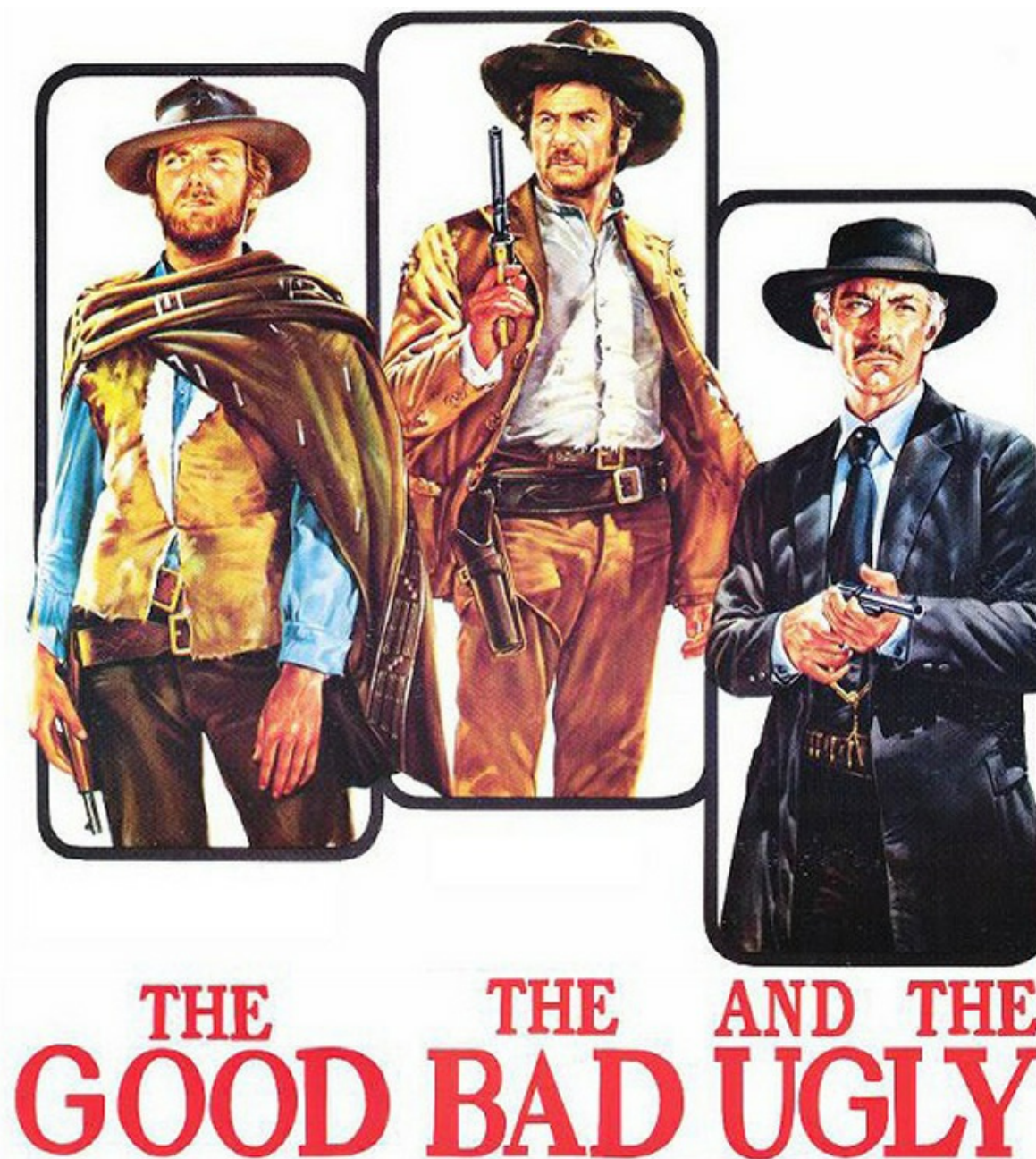
Меня зовут Сергей. Я работаю фронтенд-разработчиком в компании QIWI. В QIWI много проектов на react, redux, mobx. Но есть один, в котором используется reactive-di. Эта библиотека для управления состоянием - моя попытка ответить на вопрос "Как прекратить войну фреймворков". Т.е. зачем понадобилось изобретать еще одну библиотеку.

WTF

- material или bootstrap для каждого
- view + state coupling
- setState, redux, mobx, rxjs?
- vendor lock in
- angular, react, vue

Как может проявиться эта война, Например мы не можем сделать один универсальный material или bootstrap набор компонент. Потому что в разных фреймворках по-разному меняется состояние и этот механизм тесно связан с версткой. Нет слоя, абстрагирующего нас от redux или mobx, setState. Мы получаем vendor lock in. Берем мы angular, react, vue или что-то еще, мы сильно привязываем код приложения к этим фреймворкам.

Убийцы архитектуры



Причины этого в "убийцах архитектуры". Т.е. подходах, которые усложняют взаимозаменяемость частей приложения. Что бы было понятнее о чем я, сперва расскажу про архитектуру.

Эра монолитного ядра

- Component = view + data + logic

Сейчас большинство js-фреймворков - это обычно одно ядро, вокруг которого накручено много всего. Например, react занимается и подготовкой представления и состояние там есть, и логика вокруг него накручена. В той или иной степени аналогичная тенденция и в angular, vue, glimmer и многих других. Десятки их, если не сотни.

От ядру к слоям

- Легкий каркас, библиотека, интеграция
- Микросервисы, микроядерность, метафреймвоки, BYO
- PHP - Symfony, silex
- Java - Spring
- JS - Angular2

Если посмотреть, как развивались другие языки, например, java, php, то видно, что от монолитности постепенно переходят к концепции, когда фреймворк - это очень легкий каркас для связи множества мелких библиотек через интерфейсы. Есть сторонняя библиотека, к ней пишется слой интеграции в фреймворк и дальше она используется как его часть. Даже говорят мета-фреймворк или BYO (BUILD YOUR OWN). Например: В PHP есть symfony, а есть его облегченная версия - silex, на тех же библиотеках, в Java аналогично с Spring. У нас, на фронтенде, наиболее близок к этой концепции - angular2, за исключением того, что сторонние библиотеки переизобретены командой angular2.

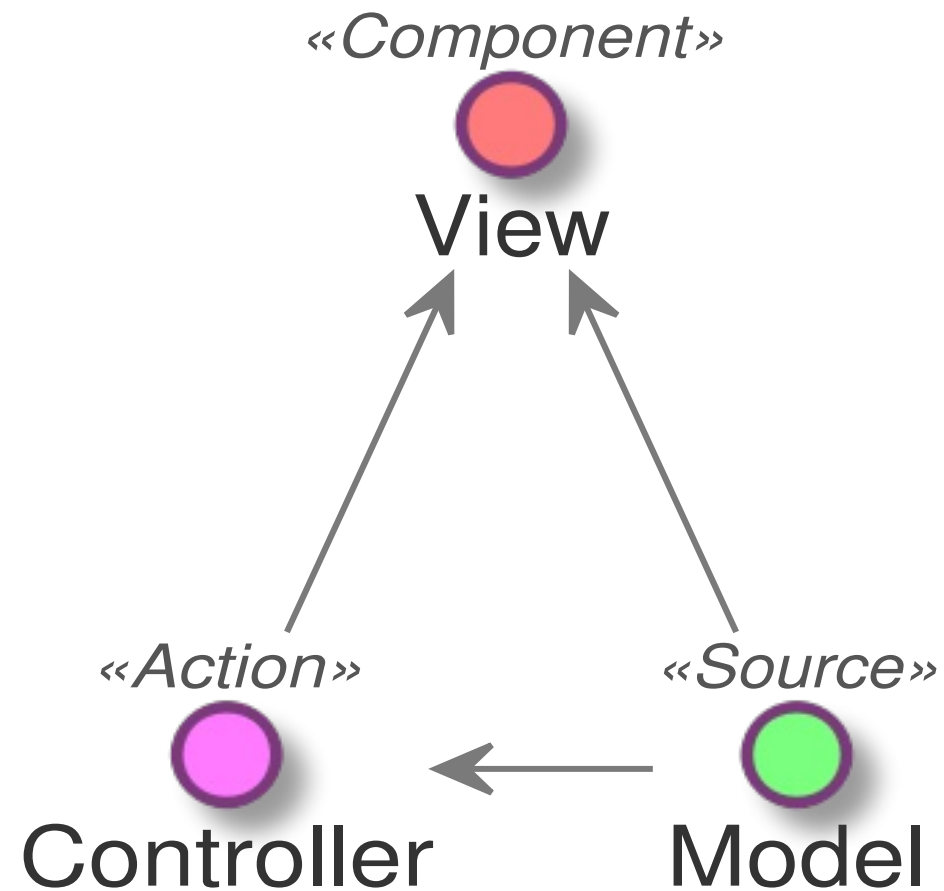
MONOLITH



MICRO SERVICES



Слои



Все части приложения можно разделить по слоям - данные, представления, логика. Если мы меняем view слой, то это затронет только его, если логику, то изменятся и представление, если структуру данных - это затронет все слои. Иными словами: что чаще меняем, то делаем менее связанным с остальными частями. Могут быть разные вариации этого подхода (FLUX например), но данные, логика и представления обычно разделяются.



```
class Some extends React.Component {  
  render() {  
    // ...  
  }  
}
```

Я упомянул связи - стрелки. Они бывают сильные: когда `extends React.Component` или `React.createElement` замаскированный под `jsx`. Этот тип связей упрощает навигацию по проекту и отладку в отсутствии мощных IDE и др. инструментов разработчика, т.к. очевидно, что это за реализация.



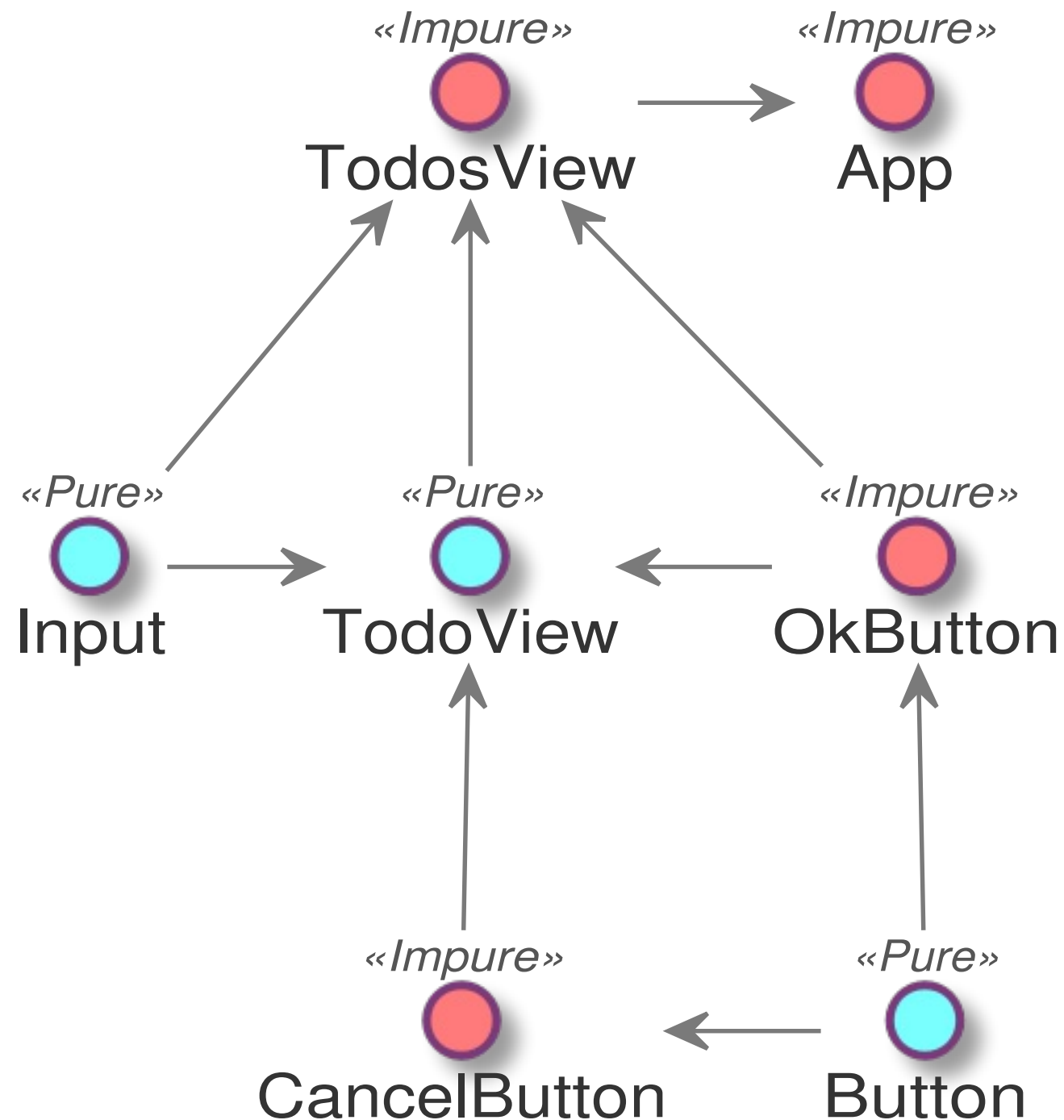
```
interface IUser {  
  name: string;  
}  
  
function MyComponent(props: { user: IUser }) {  
  return <div>{user.name}</div>  
}
```

Так и слабые: интерфейсы на props. Когда не очевидно, что за реализация IUser, она где то в другом месте задается. Для масштабирования и поддержки большого приложения важны как раз слабые связи. Проблема в том, что для работы с ними важны мощные инструменты разработки, Java+Spring+IDEA. Иначе усложняется навигация и рефакторинг такого кода. Поэтому переход к слабой связанности - это вопрос взросления экосистемы и усложнения задач.

Функции

- `f(x)`
- `f(context)(x)`
- `new F(context).method(x)`
- Инициализация - установка context

Компоненты

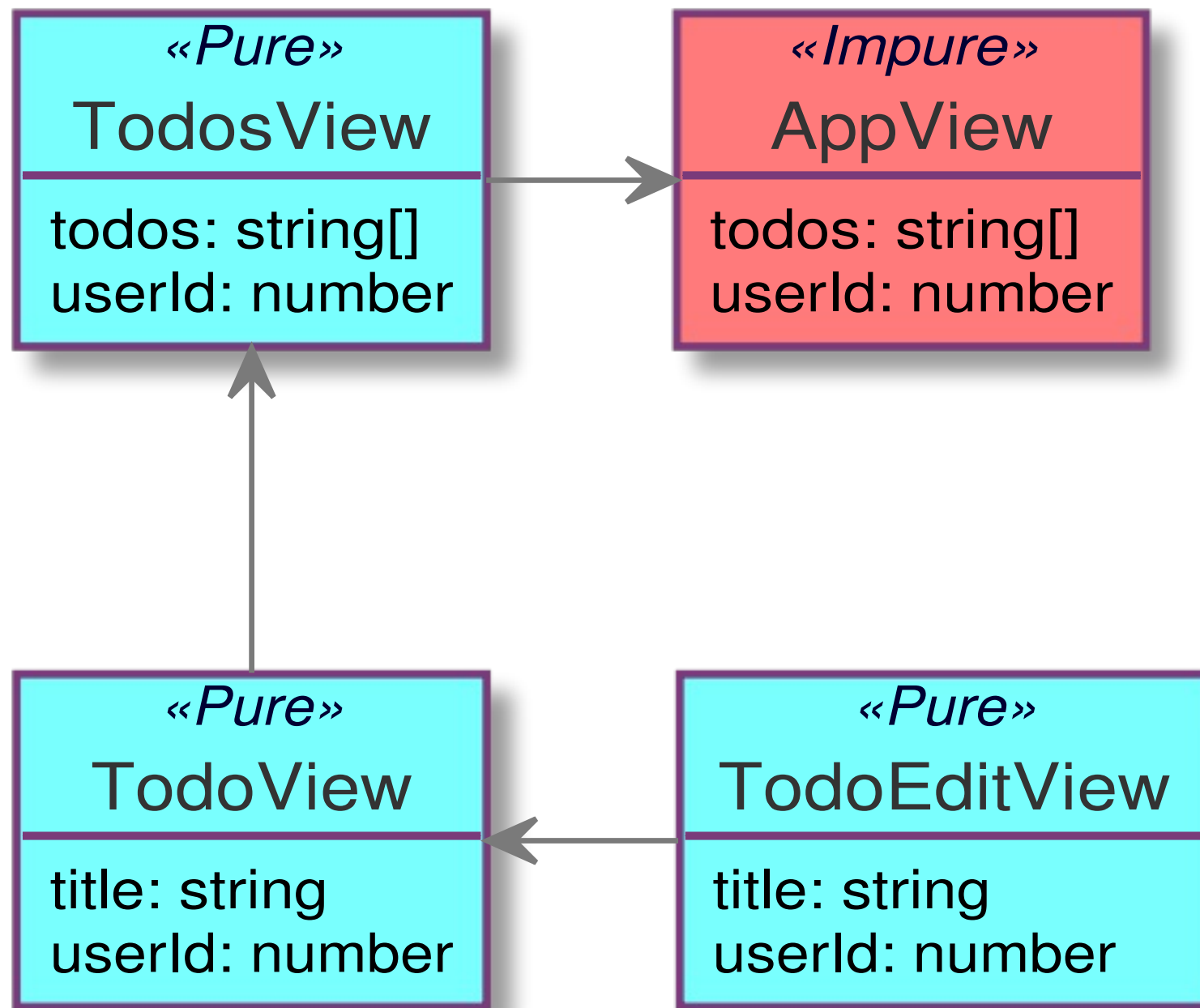


Применительно к компонентам. Все, кто программировал на реакте, знают, что компоненты бывают pure и statefull. Поведение первых зависит только от свойств, вторые от свойств и еще от контекста, под контекстом подразумевается и состояние и React.context, разница между ними только в реактивности.

ЧИСТЫЙ КОМПОНЕНТ

- Он же dumb, presentational
- `view = component(props)`
- Легкость переиспользуемости
- Рефакторинг: $O(\text{depth} * \text{props})$

Чистый компонент, он же dumb, presentational - функция от свойств (иными словами шаблон, template). Основное преимущество в том, что все или большинство ручек управления публичны, мы можем менять его поведение как угодно через них - т.е. компонент легко переиспользовать. Есть обратная сторона - сложно рефакторить приложение, по-большей части состоящее из таких компонент.



Представим, что состояние есть только в корневом компоненте страницы, а все остальное - из чистых компонентов, вот свойство `userId` в `TodoEditView` стало не нужным, в результате нам надо удалить его из всей цепочки. т.к. оно просто транзитом прокидывается вниз от `AppView`. Из-за сложности рефакторинга $O(\text{depth} * \text{props})$, в реальном приложении не бывает только чистых компонент, это и отличает фронтенд от бэкенда, иначе это был бы просто шаблонизатор.

Чистый компонент

```
function CounterView({count}) {  
  return <div>  
    Count: {count}  
  </div>  
}
```

На самом деле нет

```
function CounterView({count}) {  
  return React.createElement('div', null, 'Count: ', count)  
}
```

- vendor lock in
- Переиспользование в реакт

Но если собрать с babel-preset-react то появится прямая зависимость от React. Нельзя переиспользовать чистый компонент в другом фреймворке, поддерживающим JSX. Однако, можно продолжить мысль и переиспользовать в рамках языка и среды, т.е. уменьшить долю каркаса, постоянной части до минимально возможной.

По-настоящему чистый

```
function CounterView({count}, h: createElement) {  
  return h('div', null, 'Count: ', count)  
}
```

- Ослабить связь
- createElement как аргумент
- babel plugin для автоматизации инъекции

Для этого надо ослабить связь с createElement, например, добавив в конец аргумент, реализующий интерфейс createElement. Такой компонент можно где угодно переиспользовать, задав соответствующий h. Конечно усложняется написание компонента, надо добавлять аргумент, но это легко автоматизируется через babel плагин.

Компонент с состоянием

- Он же smart, hiorder, container
- `view = component(props)(state)`
- State - труднее переиспользовать
- Легче рефакторить - $O((\text{depth} * \text{subProps}) + \text{state})$
- `props = subProps + state`

Компонент с состоянием кастомизировать сложнее, т.к. вся логика вокруг state - это приватные детали его реализации и расширять их мы больше не можем. Заранее не всегда можно сказать, потребуется ли менять или расширять их. Но с этим мирятся, т.к. приложение, где много компонент с состоянием легче рефакторить, публичных свойств меньше - часть их перетекает в state.

```
class CounterView
  extends React.Component<void, {name: string}, {count: number}> {

  state = {count: 1}
  constructor(props: Props) { super(props) }

  render() {
    const increment = () => this.setState(({count}) => {
      count: count + 1
    })

    return <div>
      {this.props.name}: {this.state.count}
      <button onClick={increment}>Add</button>
    </div>
  }
}
```

```
class CounterView
  extends React.Component<void, {name: string}, {count: number}> {

  state = {count: 1}

  constructor(props: Props) { super(props) }
```

- Сильно связан с React.Component
- flowtype связан с React
- Конструктор занят под props

```
class CounterView
  // ...
  render() {
    const increment = () => this.setState(({count}) => {
      count: count + 1
    })
    // ...
  }
}
```

- setState
- view + state + logic

```
import counterService from './counterService'

class CounterView extends React.Component {
  // ...
  count: counterService.add(this.state.count)
  // ...
}
```

- Сиглтоны не дают подменить counterService
- Требуется рефакторинг вместо расширения

Мы прям в компоненте писали `count + 1`, а что если надо вынести эту логику в отдельный `counterService`? Можно все фигачить на синглтонах и импортах и сделать жесткую связь с ним. Но, допустим выделили компонент в стороннюю библиотеку, заиспользовали в 10 проектах, а в 11м потребовалось дополнить метод `add` логикой валидации. Тут жесткая связь начинает нам это мешать делать, приходится рефакторить, добавлять публичное свойство. Ключевой момент - масштабирование идет не через расширение, а через рефакторинг.

```
class CounterView extends React.Component {  
  // ...  
  static contextTypes = {  
    counterService: PropTypes.object  
  }  
  
  // ...  
  count: this.context.counterService.add(this.state.count)  
  // ...  
}
```

- Страшный DI
- PropTypes - эмуляция типизации
- Плохо работает в typescript

С этим можно бороться через React.context, правда он страшный, фейсбукеры сами его стыдятся, поэтому не сильно документируют. Понятно почему, такая реализация внедрения зависимостей (dependency injection), черевата увлекательным отловом багов на продакшене, вместо скучных подсказок typescript или flow на этапе написания кода. PropTypes - это эмуляция типизации, лохматое легаси со времен отсутствия flow и принятия ts. Вообще это немного напрягает, если ошибки, связанные с типами приходится отлавливать в run time, хотя рубистам нормально.



Я тут все про реакт, да про реакт, а что с другими... Vue, aurelia, glimmer, deku, yo-yo, cyclejs, funnel... Стоит пожалуй только angular2 упомянуть, т.к. среди всего этого зоопарка только он чуть приподнялся на ступеньку.

Angular2 + контекст

```
@Component ({
  selector: 'my-counter',
  templateUrl: './counter.component.html'
})
class CounterView {
  counterService: CounterService
  counter: number = 0
  @Input name: string

  constructor(counterService: CounterService) {
    this.counterService = counterService
  }

  addCounter() {
    this.counter = this.counterService.add(this.counter)
  }
}
```

```
// ...  
class CounterView {  
    // ...  
}  
CounterView._deps = [CounterService]
```

- typescript + reflection = metadata
- Сыро и без спецификаций

Что бы магия заработала, ангуларовцы слегка прогнув микрософт с их тайпскриптом, записывают ссылку на CounterService в класс CounterView. Такая наколеночная страшенькая рефлексия без всяких спецификаций, привнесенная в typescript. А среда исполнения, т.е. dependency injection ангулара, по этой ссылке подсовывает готовый объект.

```

@Component ({
  selector: 'my-counter',
  templateUrl: './counter.component.html'
})
class CounterView {
  counterService: CounterService
  counter: number = 0
  @Input name: string

  addCounter() {
    this.counter = this.counter + 1
  }
}

```

- Component = template + interface + state + logic
- View model - управляет реактивностью
- Как прикрутить mobx, не поверх, а вместо
- changeDetection - часть монолита

Один к одному сцепили шаблон, описание контракта к этому шаблону, модель, и логику по работе с ней. Сам класс компонента ангулара это View Model. Однако в ней слишком много ответственности: она управляет рендерингом компонента и эту логику никак нельзя отделить. Нельзя прикрутить mobx, вместо, а не поверх. Нельзя заменить changeDetection на свой, что может потребоваться, когда(не если, а когда) приложение на angular начнет тормозить.



Vendor lockin everywhere

Переизобретение и копипаст

- `redux/vuex`
- `glimmer = component + mobx`
- `deku = component + redux`
- `{React, Vue, Inferno, Redux, Mobx}-router`
- `inferno-{redux, mobx}`

Монолитные анализаторы

- JSX + flow = контракт к шаблонам
- Angular2 templates != typescript
- Прибить flow к React.Component
- Типы и JSX в Vue, Deku?

Ребята из фейсбука придумали jsx и flow, который понимает этот самый jsx, относительно компонент с состоянием - flow просто прибит гвоздями к реакту. Там в коде есть прямая зависимость анализатора от extends React.Component. А что делать ребятам из deku, vue? Переизобретать свой flow, как они это сделали с redux (vuex)? А typescript еще более монолитный - это не только анализатор, а еще компилятор, для которого до недавнего времени нельзя было даже свой transformation-plugin написать. Например поддержка шаблонов angular2 в ts просто прибита гвоздями по аналогии с jsx, компилятор знает о фреймворке.

Оптимизация = конкуренция



- Хайп $5 > 3$
- react fiber, vdom, prepack, inferno
- Не имеет отношения к решению

Про оптимизацию слишком много хайпа, в основном, все современные тенденции во фронтенде это про то, кто больше попугаев покажет в ui-bench: fiber, vdom, prepack, inferno. Оптимизация нужна из-за отставания браузеров от бизнес задач и медленной скорости их развития из-за легаси из которого состоит web. Так проще конкурировать, цифрами убедить проще, т.к. меньше надо знать. React 3 попугая выдает, Inferno 5, значит Inferno лучше. Конкурировать, доказывая архитектурные преимущества, гораздо сложнее. Т.к. проявляются эти преимущества не сразу и на достаточно больших задачах, увидеть их можно только в сравнении, пройдя опыт и говнокодной разработки.

Оптимизации в приложении

```
class CounterView extends React.Component {
  state = {count: 0}

  shouldComponentUpdate(nextProps, nextState) {
    return nextState.count === this.state.count
  }

  _add = () => {
    this.state.count++
    this.forceUpdate()
  }

  render() {
    return <div>{this.props.name}: {this.state.count}
      <button onClick={this._add}>Add</button>
    </div>
  }
}
```

Angular

```
@Component({
  template: '{{counter}}',
  changeDetection: ChangeDetectionStrategy.OnPush
})
class CounterView {
  counter = 0

  constructor(private cd: ChangeDetectorRef) {}

  add() {
    this.counter = this.counter + 1
    this.cd.markForCheck()
  }
}
```

- Event -> viewRef.detectChanges
- Minesweeper
- OnPush = shouldComponentUpdate

Думаете в angular2 лучше? Там на любое событие дергается detectChanges. Это видимо тормозной на больших приложениях механизм, который правильнее было бы не делать в ангуларе вовсе, а вынести в стороннее решение. Тут changeDetection.OnPush такой же костыль как и shouldComponentUpdate.



Это я к тому, что оптимизация в коде приложения не нормальное явление, как нам пытаются преподнести из многочисленных маркетинговых докладов. Это признание несостоятельности идеи или реализации фреймворка касательно автоматической оптимизации. Идея определяет резервы оптимизации, а хорошие алгоритмы только помогут полнее их реализовать, как в Inferno.

Mobx

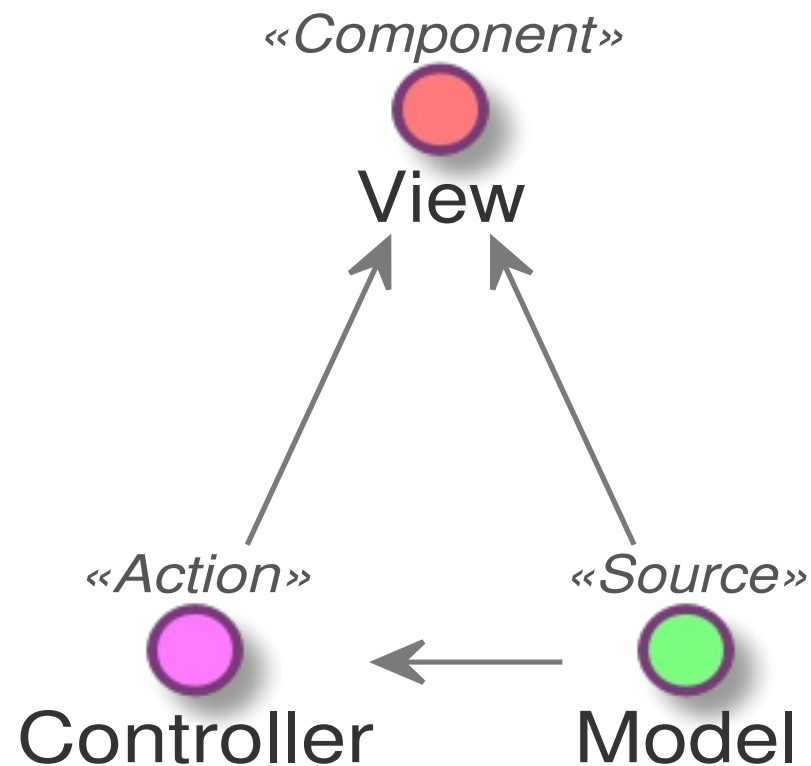
```
const CounterView = observer(store => <div>{store.count}</div>)  
  
const AppView = observer(store => <div>  
  <CounterView count={store}/>  
</div>)  
  
class Store {  
  @observable count: number = 0  
}  
  
<AppView store={new Store()} />
```

В свете оптимизации стоит упомянуть mobx. Особенность его в том, что при помощи магии get/set, компоненты подписываются непосредственно на те свойства, которые они используют в Store. Можно все компоненты сделать observer-ами, но только CounterView обращается к store.count, поэтому при изменении count, будет перерисован только он. Эта идея дает гораздо больше резервов оптимизации.

Mobx

- cellx, derivablejs, glimmer, mol
- Обратился к свойству - подписался
- Ранняя оптимизация
- Точность без VDOM и changeDetection
- Проблема unboxing

Это все реализации ненавязчивых стримов (derivable, cellx, mol_atom). Подписка компонента на изменения в данных происходит в момент обращения к свойствам. Оптимизация происходит раньше, в слое данных, а не в VDOM (react) или в компонентах (angular). В подобных решениях VDOM не нужен. Правда может возникнуть проблема unboxing, - подписка происходит там, где извлекаются данные из стрима.



- React - View + setState
- Mobx - Model + ненавязчивые стримы
- Reactive-di - Relations, все внутри стримов

Model

```
class Counter {  
  count: number = 0  
}
```

- State + selector + interface
- Нет undefined багов
- Нет декораторов

Вот, например, как выглядит модель. Начальное состояние и контракт в одном флаконе. При таком подходе не может быть undefined-багов, не нужно дополнительных сериализаторов/десериализаторов. На основе моделей также делаются локализации, реактивные стили через jss. Не содержит декораторов или любых других зависимостей от фреймворка.

View

```
function CounterView(  
  // public:  
  {name}: { name: string; },  
  
  // private:  
  {counter, actions}: {  
    counter: Counter;  
    actions: CounterActions;  
  }  
) {  
  return <div>  
    {name} {counter.count}  
    <button onClick={actions.add}>Add</button>  
  </div>  
}
```

Action

```
@actions class CounterActions {  
  _counter: Counter  
  
  constructor(counter: Counter) {  
    this._counter = counter  
  }  
  
  add() {  
    src(this._counter).set({  
      count: this._counter.count++  
    })  
  }  
}
```

Класс, который предоставляют компоненту методы, по сути экшены, меняющие состояние: тут может быть валидация, запрос на сервер и т.д. В отличие от традиционного DI, зависимость может быть от данных (в ангуларе это называется ValueProvider), только здесь это value реактивно. При выполнении метода add, CounterActions переинициализируется с новым значением, т.е. reactive-di это стримы, упрятанные внутрь di.

- DI для связей: data - ui - business logic
- Ненавязчивые потоки (mobx)
- Экосистема: типы, поддержка в ide: JSX
- Высокоуровневые интерфейсы: ng-modules
- Баласировать между KISS и di-окружением

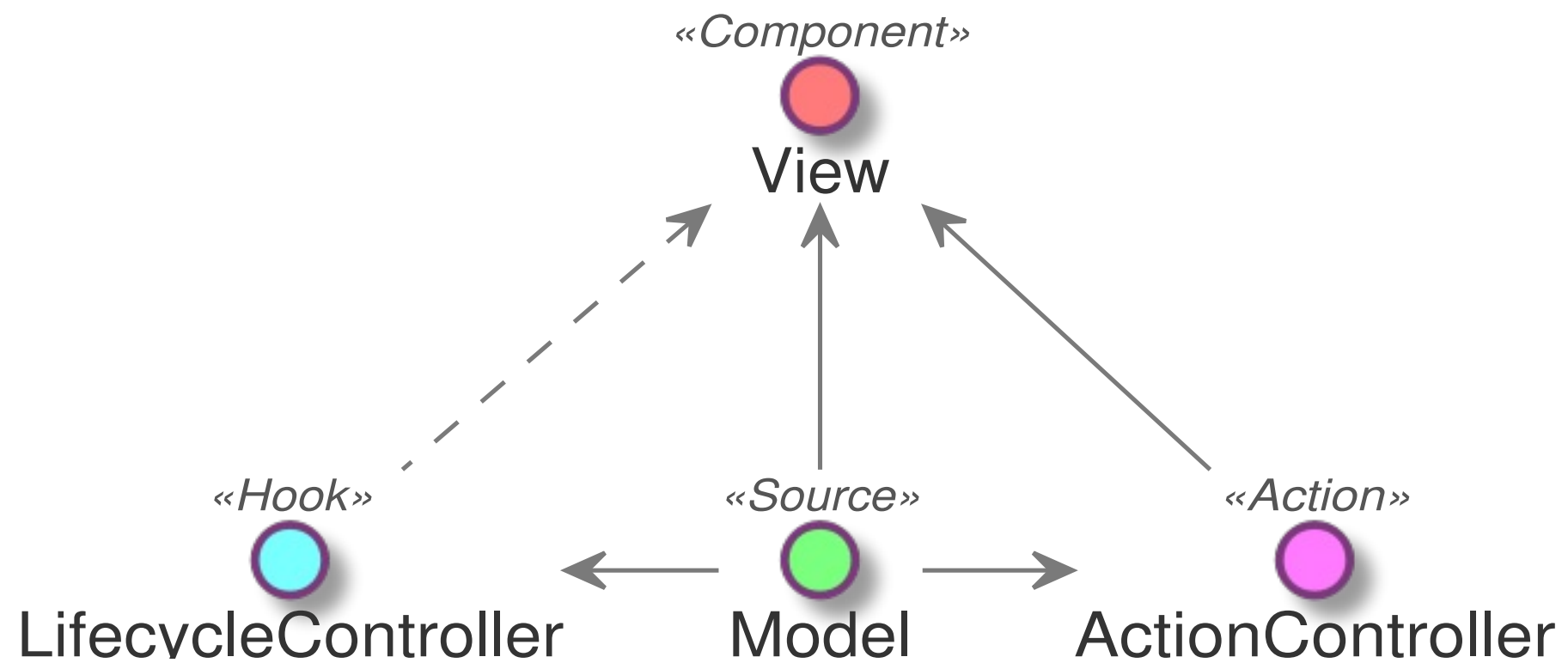
Я попытался выделить 1. DI - это ключ. Фреймворк должен быть изолированным набором библиотек, связанных через инверсию зависимостей, каждая библиотека со своей маленькой ответственностью (никаких state и context в ui). 2. Проектируя фреймворк, думать в первую очередь о типах и безопасности на всех уровнях MVC. 3. Должна быть настоящая модульность, центральное состояние, как в redux, не годится. 4. Фреймворк должен помогать строить приложение из иерархически выстроенных мини-приложений, которые интегрируются в общую шину, ng-modules хороший пример, аналогов которому я не нашел в мире react. 5. Оптимизация в слое фреймворка, а не приложения. Стремиться делать фреймворк таким, что бы упрощался рефакторинг. Оставаться простым и искать компромис между хилыми возможностями js/babel/flow/typescript платформ и быть дружелюбным к DI.



Путь тернист и конца пока не видно. Как же прекратить войну фреймворков? Осознание проблем - это уже много. Надеюсь я смог показать, что в нашей любимой фронтенд архитектуре есть проблемы, которые не заметны с близкого расстояния, но видны на больших. На мой взгляд, будущее за развитием `mobx`, за развитием и упрощением `angular2`, они здесь делают первые шаги. Подобные идеи заслуживают более яркого освещения среди нас, программистов. Я, в свою очередь, некоторые идеи реализовываю в `reactive-di`.

- github.com/zerkalica/reactive-di

- Идеального инструмента нет
- Но будет в ближайшие годы
- Angular2 - шаг вперед
- Однако, не KISS
- Не сбалансировали сложность и качество
- Избегайте хайпа



redux

- Шаблонный код
- Сложнее менять состояние
- Центральное состояние
- Типизация требует еще больше шаблонов
- Простая база, но бесполезная без сложного окружения

Вот так плавно мы перешли к библиотекам управления состоянием. Например, redux это не про то, как уменьшить кол-во шаблонного кода, не про то, как проще поменять состояние, не про то, как разделить состояние на много кусочков т.к. основная идея - это центральный стейт. Не про то, как писать с опорой на типизацию. Это все можно конечно сделать, но путем дополнительных усилий, в виде redux-thunk, action-creator-ов, saga т.п. решений.

```

export type Action =
  { type: 'LOADED_ABOUT', list: Array<ParseObject> }
| { type: 'LOADED_NOTIFICATIONS', list: Array<ParseObject> }
| { type: 'LOADED_MAPS', list: Array<ParseObject> }
| { type: 'LOADED_FRIENDS_SCHEDULES', list: Array<{ id: string; name: string }> }
| { type: 'LOADED_CONFIG', config: ParseObject }
| { type: 'LOADED_SESSIONS', list: Array<ParseObject> }
| { type: 'LOADED_SURVEYS', list: Array<Object> }
| { type: 'SUBMITTED_SURVEY_ANSWERS', id: string; }
| { type: 'LOGGED_IN', source: ?string; data: { id: string; name: string; } }
| { type: 'RESTORED_SCHEDULE', list: Array<ParseObject> }
| { type: 'SKIPPED_LOGIN' }
| { type: 'LOGGED_OUT' }
| { type: 'SET_SHARING', enabled: boolean }
// ...

export type Dispatch = (action: Action | ThunkAction | PromiseAction) => void;
export type GetState = () => Object;
export type ThunkAction = (dispatch: Dispatch, getState: GetState) => Action;
export type PromiseAction = Promise<Action>;

```

Типизацию в js не любят. Однако, если попробовать честно описать интерфейс диспетчера, то надо сперва описать общий экшен как union-тип, всех экшенов в приложении (если используется один экземпляр dispatch). Это несколько затрудняет разбиение на модули. Это из-за одного диспетчера и единого стейта.

```
// ...  
// single state - less modularity  
function mapStateToProps(state: {user: UserState}) {  
  return { name: state.user.name } // state.user undefined  
}  
const AppContainer = connect(mapStateToProps)(CounterView)
```

- Single state
- Типы состояния и редьюсеров

Рассмотрим react-redux (как и в mobx-react). Во первых, несмотря на интерфейс, state в mapStateToProps приходит от всего приложения, т.е. центральное состояние нарушает модульность приложения. Во вторых, нет сопоставления типов состояния и редьюсеров в combineReducers. state.user undefined, т.к.

```
// reducer is not type checked  
const reducer = combineReducers({ xyz: user })  
const store = createStore(reducer)
```

- flowtype не отловит xyz вместо user

```
// store in Provider is not type checked
// How to pass something else
// Not here
<Provider store={null /* store */}>
  <CounterContainer title="123" />
</Provider>
// flow check: Found 0 errors
```

- Provider store interface
- Только store
- Прибито к React вместо DI

```
function CaseComponent({history}) {  
  return <Router history={history}>  
    <Route path="/" component={App}>  
      <Route path="foo" component={Foo} />  
      <Route path="bar" component={Bar} />  
    </Route>  
  </Router>  
}
```

- ReactRouter, ReactSideEffect, ReactHelmet
- Контроллеры в слое с шаблонами


```
class RouterState {
  path: string
}

function CaseComponent (
  {routerState}: {
    routerState: RouterState
  }
) {
  switch (routerState.path) {
    case '/': return App
    case 'foo': return Foo
    default: return App
  }
}
```

А ведь достаточно просто развязать это все через состояние. Строка браузера влияет на состояние, например `mobx`, а дальше делается `CaseComponent`, который уже выбирает нужный. И не надо прибавлять роутинг к реакту, а потом делать убыстренный клон реакта `inferno`, и коипастиь его туда, как с `inferno-router`.

- `shouldComponentUpdate` - костыль
- React не проектировали
- Сперва `mobx`
- `VDOM`, `setState`, `router` не нужны
- `Redux` - прослойка к `mobx`, а не к `component`

Костыль этот существует потому, что `react` начали проектировать с конца, с `view` слоя, а не со слоя данных. Если бы сперва слепили `mobx`, а на его основе сделали `react`, на большую часть фигни просто бы не тратили ресурсы: `VDOM`, `setState`, `router`, все бы упростилось до нельзя. А `redux` - стал бы прослойкой к `mobx`, что позволило бы изежать центрального стейта.

Lifecycle

```
@hooks (Counter)
class CounterHooks {
  pull(counter: Counter): Observable<Counter> {

    let count = counter.count

    return new Observable((observer: Observer<Count>) => {
      setTimeout(() => observer.next(++count), 1000)
    })
  }
}
```

Часто бывает так, компонент отрендерился и вам нужно актуализировать его состояние. Тут помогают механизмы, которые есть в некоторых ORM на других языках (Doctrine, Hibernate). Логика актуализации состояния Counter задается в таком сервисе. Когда первый раз отрендерится хотя бы один компонент, использующий Counter, выполнится метод pull и Observable с этого момента будет управлять Counter ом. В mobx аналогично сделан хелпер where, в cellx и mol есть похожие механизмы.