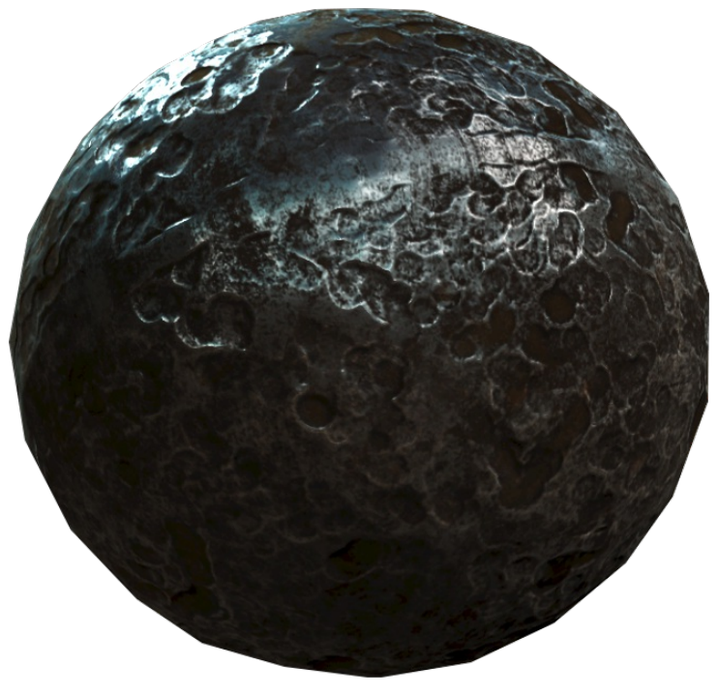
От фреймворков к сверхфреймворкам

Меня зовут Сергей. Я работаю фронтенд-разработчиком в компании QIWI. Вообще фронтенд разработкой занимаюсь примерно 5 лет. До этого примерно столько же писал на php. Меня всегда интересовало как писать безопасный и переиспользуемый код. Я расскажу о проблеммах фреймворков, которые мешают нам это делать. Цель - развить идею сверхфреймворка, который слабо связан с кодом приложения и является посредником для различных сторонних библиотек.

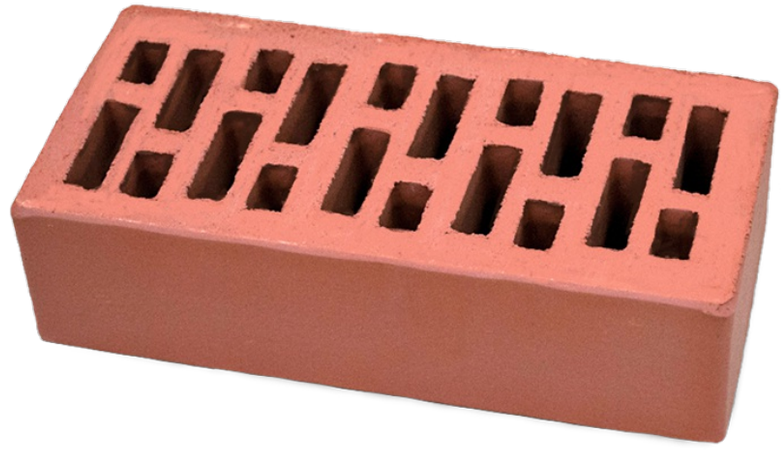
- PHP - Symfony, silex
- Легкий каркас, библиотека, интеграция
- Типы, контракты
- Микросервисы, микроядерность

Если посмотреть, как развивались другие языки, например, java, php, то видно, что от монолитности постепенно переходят к концепции, когда фремворк - 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. 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.



- Component = view + data + logic
- React.setState, redux, rxjs, mobx?
- ts, flow (Angular2 driven)

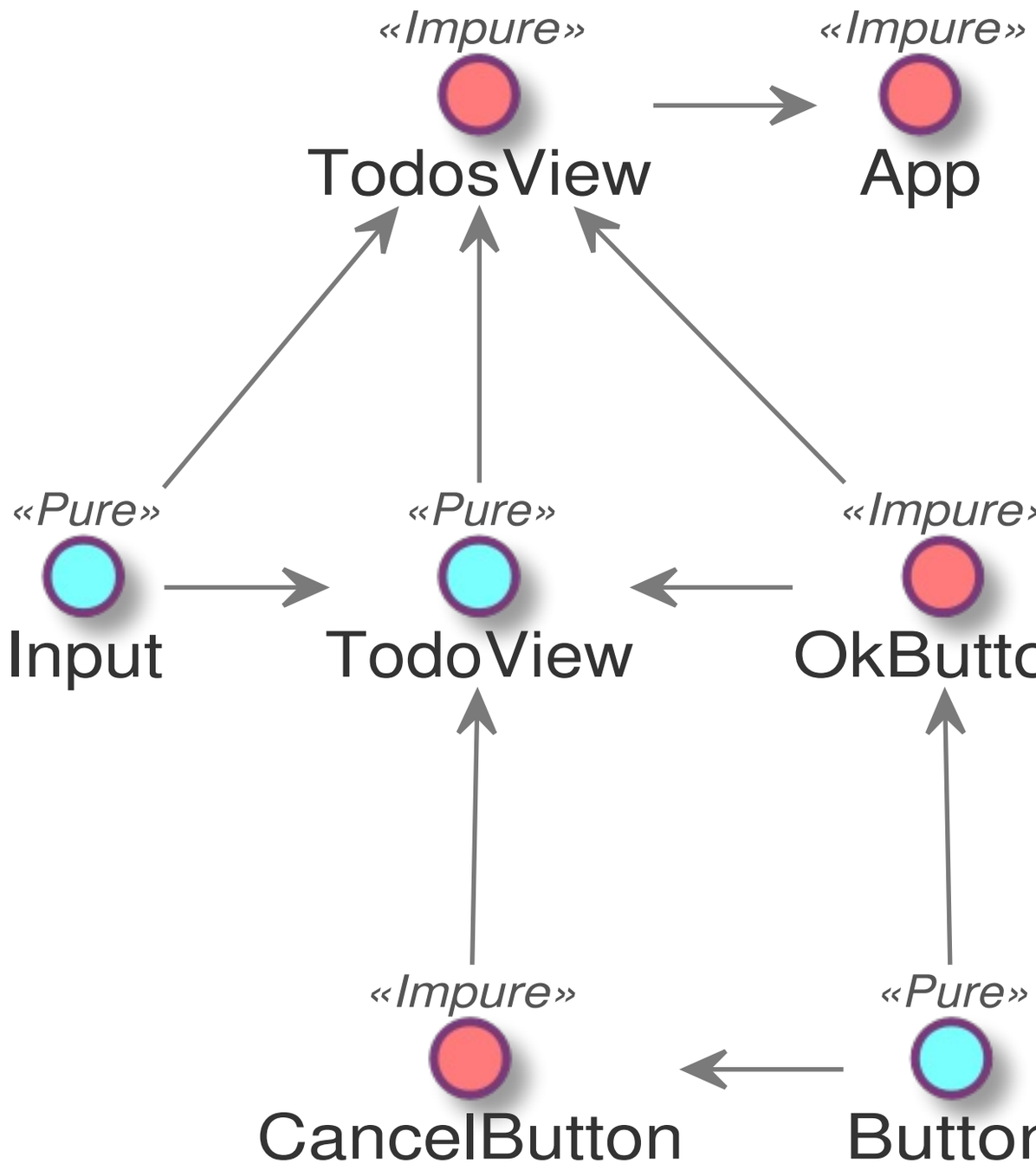
Сейчас большинство js-фреймворков - это обычно одно ядро, вокруг которого накручено много всего. Например, react занимается и подготовкой представления и состояние там есть, и логика вокруг него накручена.



- f(props)
- f(context)(props)
- new F(context).method(props)

Базовый кирпич - это функция. Не важно о чем речь, шаблон или код. Сами функции бывают как чистые, так и с некоторым контекстом. Классы - это набор функций с контекстом в виде this.

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.

Context

- Vue slot
- Constructor, HDI в Angular2
- React.context / Provider

React

- Presentional (view)
- No framework reuse
- Container (injector, view)
- No app reuse

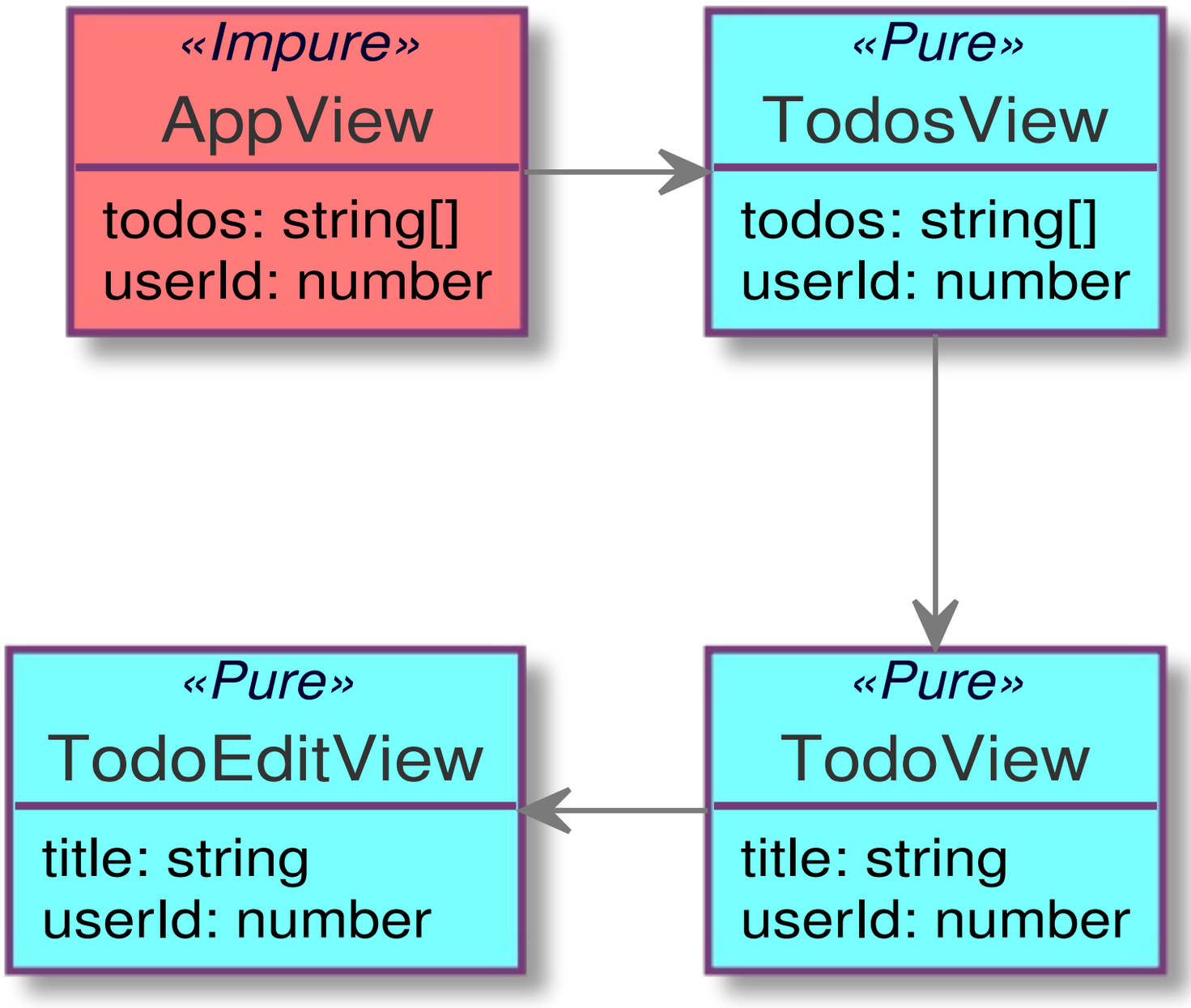
Есть dumb-компоненты (больше view), казалось бы что проще, а есть smart (больше контроллер).

Чистый компонент

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

- JSX + flow = контракт к шаблонам
- Кастомизируемость
- Рефакторинг: O(depth * 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 обротная 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 компонент, 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(depth * props), в реальном приложении не бывает только чистых компонент, это и отличает фронтенд от бэкенда, иначе это был бы просто шаблонизатор.

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

- чистый компонент != чистая функция
- ослабить связь

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.

vue-jsx

```
Vue.component('jsx-example', {
  render (h) { // <-- h must be in scope
    return <div id="foo">bar</div>
  }
})
```

h auto-injection

```
Vue.component('jsx-example', {
  render () {
    // const h = this.$createElement
    return <div id="foo">bar</div>
  }
})
```

- Зависимость от Vue.component

Нуль-компонент

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

h auto-injection

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

Переиспользовать

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

Компонент с состоянием

- view = component(state)(props)
- state - труднее кастомизировать
- O((depth * subProps) + state)
- props = subProps + state

Компонент с состянием 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) }

  add() {
    this.setState({ count: this.count++ })
  }

  render() { /* ... */ }
```

- React.Component
- Конструктор занят под props
- setState

```
import Component from 'my-react-like'
```

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

  some: Some

  constructor(some: Some) { super(); this.some = some }

  render() { /* ... */ }
}
//  ...
```

```
<CounterView name={123} /> // 0 errors
```

Типы и JSX в Vue, Deku?

```
function CounterView(props: {count: number, add: () => void}) {
  return <div>
    {props.count}: <button onclick="{add}">Add</button>
  </div>
}
```

```
function mapStateToProps(store) {
  return { count: store.counter.count }
}
const CounterContainer = connect(mapStateToProps)(CounterView)
```

```
<Provider store={'XYZ'}> // unsafe
  <CounterContainer/>
</Provider>
```

Например, как в redux. Оборачивают чистый компонент в connect, а в точке входа провайдят store, через Provider. Почему в свойство стор = XYZ в последнем блоке? Потому что flow и ts не могут обнаружить несоотвествие типов с тем, что в mapStateToProps. Как работает Provider внутри?

```
class App extends React.Component {
  static childContextTypes = {
    store: PropTypes.object
  }

  getChildContext() {
    return { store: this.props.store }
  }

  render = () => <CounterContainer/>
}
```

```
class CounterContainer extends React.Component {
  static contextTypes = {
    store: PropTypes.object
  }

  render = () => CounterView({ count: this.context.store.count })
}
```

Есть механизм React.context. В App мы регистрируем зависимости через getChildContext и childContextTypes. В CounterContainer мы вытаскиваем данные из контекста. Механизм этот страшный, фейсбуковцы сами его стыдятся, поэтому не сильно документируют. PropTypes - это эмуляция типизации, лохматое легаси со времен отсутствия flow. Такое решение не может нормально интегрироваться в ts или flow.

Вообще, ui-фреймворков очень много, я не буду всех их упоминать. Сказать стоит пожалуй только про angular2, т.к. несмотря на свои недостатки, он среди всего этого зоопарка чуть приподнялся на ступеньку.

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

  constructor(private counterService: CounterService) {}

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

- Component = template + view model + logic
- PropTypes на constructor

Angular2: Один к одному сцепили шаблон, описание контракта к этому шаблону, модель, и логику по работе с ней. На ней слишком много ответственности. Нельзя прикрутить mobx, вместо, а не поверх changeDetection. Нельзя заменить changeDetection на свой, что может потребоваться как ради экспериментов, так и ради оптимизаций. Ребята из команды angular2 идею контекста сделали центральной. В итоге это гораздо ближе к нативному синтаксису typescript.

```
const Injectable = 0 as any

interface ITest {}
class CounterService {}

@Injectable()
class CounterView {
  constructor(private cs: CounterService, test: ITest) {}
}
```

tsc --emitDecoratorMetadata test.ts

```
Reflect.metadata(CounterView, "design:paramtypes", [
  CounterService,
  Object
])
```

ITest -> Object, WAT?

map[ITest] = SomeClass

Что бы магия заработала, ангуларовцы, слегка прогнув микрософт с их тайпскриптом, записывают сигнатуру конструктора в метаданные. Dependency injection ангулара, вместо CounterService подсовывает готовый объект. Это называется рефлексия, во многих языках она из коробки, в ts прибитая к декораторам и не работающая с интерфейсами. Например, итерфейсы просто заменяются на Object. map[ITest] = SomeClass можно делать в C# и Dart, однако в дартовом ангуларе не используется эта фича, в отличие от C# Ninject. Именно из-за слабого развития инструментов и типизации, позволяющих делать reflection, DI был так непопулярен у нас на фронтенде. Поэтому аналогично с типами в JSX-шаблонах у ангулар2 нет здесь конкурентов.

Angular2 templates

```
@Component({
  selector: 'app',
  template: `{{cnt}} <button (click)="addSome()">Add</button>`
})
export class CounterView {
  counter: number = 0
  add(){
    this.counter += 1
  }
}
```

- Типы в шаблонах
- typescript проигнорирует addSome

Vue

```
var app5 = new Vue({
  el: '#app-5',
  data: {
    message: 'Hello Vue.js!'
  },
  mixins: [myMixin],
  methods: {
    reverseMessage: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
```

- К React.createClass, опять?
- fuck the flow
- Быть всем
- Быть всем в монолите

vue-property-decorator

```
import { Component, Inject,
  Model, Prop, Vue, Watch } from 'vue-property-decorator'

@Component class MyComponent extends Vue {
  @Inject() foo: string

  @Model('change') checked: boolean

  @Prop({ default: 'default value' }) propB: string

  @Prop([String, Boolean]) propC: string | boolean

  @Provide() foo = 'foo'

  @Provide('bar') baz = 'bar'

  @Watch('child')
  onChildChanged(val: string, oldVal: string) { }
}
```

vuex - vue only

use with react #550

Closed **weepy** opened this issue on 30 Dec 2016 · 1 comment

weepy commented on 30 Dec 2016

is it possible to use Vuex with React ?

ktsn commented on 30 Dec 2016 Member

As Vuex is well optimized for Vue.js, we cannot use Vuex without Vue.js. So you should not use it with React.

ktsn closed this on 30 Dec 2016

В продолжении темы монолитов следует сказать про копипаст. Я уже говорил про универсальный каркас, куда интегрируются сторонние либы. Так вот на фронтенде его нет, каждый переизобретает этот каркас в своем ядре. Причем мыслят старыми категориями безтипового js, без DI. Поэтому vuex работает только с vue.

- react-router
- react-router-redux
- mobx-react-router
- inferno-router
- vue-router
- vuex-router-sync

Роутеры переизобретаются в разных сочетаниях тоже по причине не продуманности базовых вещей. Они делают все одно и тоже, в зависимости от пути в строке бразурера отдают компонент. А копипастить код приходится из-за отсутствия универсальых интерфейсов интеграции этих библиотек с состоянием.

```
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
- Контроллер
- Смешение слоев

Таких примеров много, ReactRouter, ReactSideEffect, ReactHelmet - это все реализация контроллеров в слое с шаблонами.

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

```
class Router {
  @observable path = ''
}
const router = new Router()
location.onChange((path: string) => {
  router.path = path
})
```

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 прибивать роутинг к реакту, а потом делать убыстренный клон реакта inferno, и копипастить его туда, как с inferno-router.

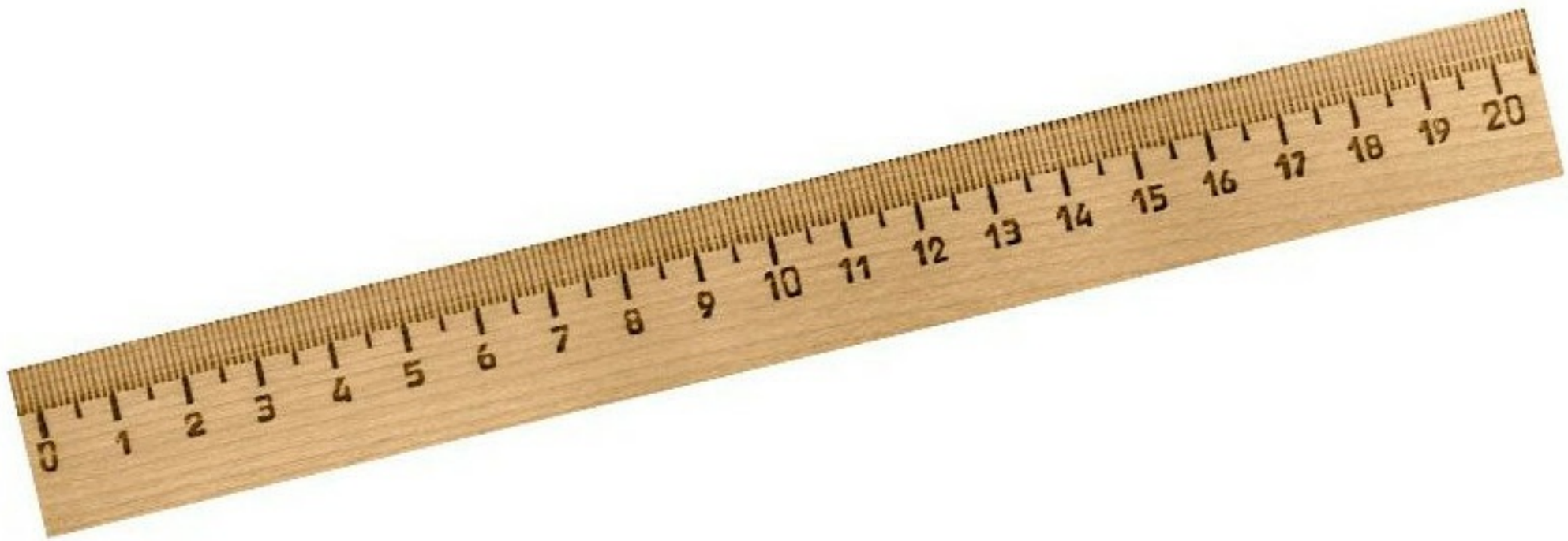
Vendor lock-in everywhere

Механизма обмена решениями между фреймворками нет. Выбрав один путь - придется и выбрать экосистему вокруг фреймворка. Конкрурировать в этой гонке могут только те, у кого больше ресурсов для хайпа.

Конкуренция

- Типовой код (angular - 15K, inferno - 5K)
- Монолитный код
- Подсадить на фреймворк
- Одиночки в худшем положении

Оптимизация фреймворков = хайп



- Хайп 5 > 3
- ~~Связанность, сцепленность~~
- 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 = костыли

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

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

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

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

Оптимизация в приложении - это доп. код, который может содержать логические ошибки и анализаторы не помогут их отловить. А вы нашли тут багу, не? Вот flow не нашел.

Angular

```
@Component({
  selector: 'app',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `{{counter}} <button (click)="add()">Add</button>`
})
export class CounterView {
  public counter : number = 0;
  constructor(private cd: ChangeDetectorRef) {}

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

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

Думаете в angular2 лучше? Там на любое событие дерагется 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. Кто-нибудь помнит, как нам несколько лет назад был хайп о том, что VDOM в реакте вообще позволит не парится об оптимизации, все сделает за вас.

Mobx

- cellx, derivablejs, mol
- Обратился к свойству - подписался
- Ранняя точная оптимизация без VDOM

В свете оптимизации стоит упомянуть mobx и идейно похожие решения - derivable, cellx, mol_atom. Это все реализации ненавязчивых стримов. 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 решения VDOM не нужен.

```
const CounterView = observer(store => <div>{store.count}</div>)

const AppView = observer(store => <div>
  <CounterView count={store}/>
</div>)

class Store {
  @observable count: number = 0
}

const store = new Store()
React.render(<AppView store={store} />, document.body)

store.count = 1 // rerender
```

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.

```
const CounterView = /*observer*/(store => <div>{store.count}</div>)

const AppView = /*observer*/(store => <div>
  <CounterView count={store}/>
</div>)

class Store {
  /*@observable*/ count: number = 0
}

const store = new Store()
React.render(<AppView store={store} />, document.body)

store.count = 1 // rerender
```

Фреймворк - это каркас, с точками расширения, куда мы вставляем данные, логику и верстку, а что если уменьшить долю каркаса до 0? Зависимость от React и mobx перейдет в зависимость от спецификации и подхода к разработке. Это позволит быть менее зависимым от хайпа.

Reactive-di

View

```
class Counter { count = 0 }

function Hello(
    // public
    {text}: { text: string; },

    // private
    {counter}: { counter: Counter; }
) {
    return <div>
        <h1>{text} {counter.count}</h1>
    </div>
}
```

Чистая верстка, с контрактом и разделением на публичный интрефейс (props) и приватный (context). Также, никаких декораторов и зависимостей от фреймворков.

```
function Counter() { this.count = 0 }

function Hello(_ref, _ref2, _t) {
    var text = _ref.text;
    var counter = _ref2.counter;

    return _t.h(2, 'div', null, [
      _t.h(2, 'h1', null, ['count ', counter.count])
    ]);
}

Hello._isComponent = true;
Hello._dependencies = [{ counter: Counter }];
```

context = DI + metadata

С помощью babel-плагина к компоненту добавляются метаданные. По метаданным движок reactive-di отличает свои компоненты от реактовых, решается проблема легаси.

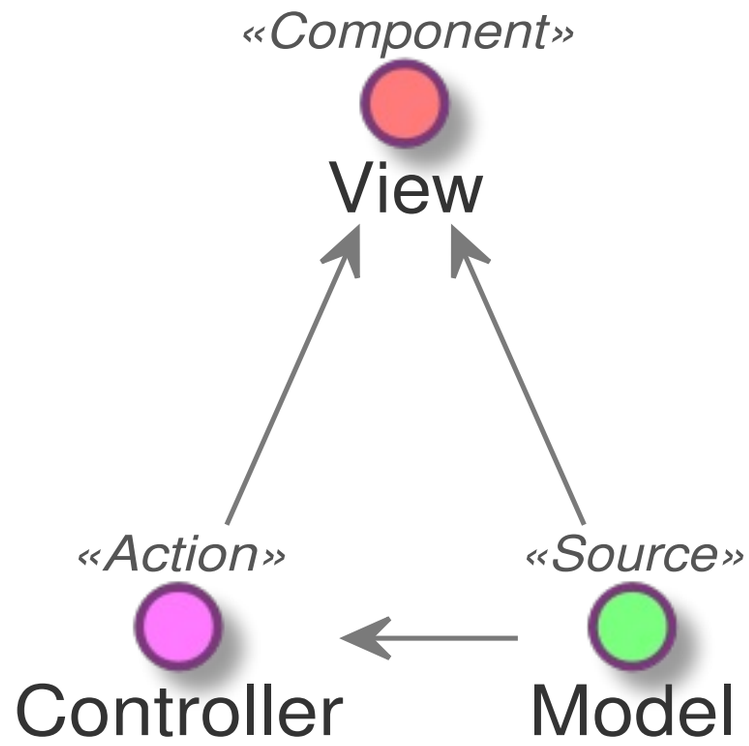
Lifecycle

```
class Counter { count = 0 }

@hooks(Counter)
class CounterHooks {

  constructor(private fetcher: Fetcher) {}

  pull(counter: Counter): Observable<Counter> {
    return this.fetcher.fetch('/api/some')
  }
}
```

- Mobx / where
- Cellx / pull

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.

- 15й стандарт
- Совместим с 14м (React)
- Поддерживается в flow
- Работает legacy
- Interoperability
- Ъ-Чистые
- ~~Smart, dumb~~



- React - View
- Mobx - Model
- Reactive-di - Окружение, все внутри стримов

reactive-di: вместо зависимостей от библиотек мы получаем зависимости от спецификаций. Фреймворк переносится из кода приложения в среду исполнения.

!!!

- Экосистема вокруг типов
- Слои: data - ui - business logic
- Ненавязчивость (mobx)
- KISS
- КПД: 3-4 (angular - 15K, inferno - 5K)

Через тернии к звездам. Идеального решения пока нет. Надеюсь я смог показать, что в нашей любимой фронденд архитектуре есть проблемы, которые не заметны с близкого расстояния, но видны на большом. Идеи ненавязчивых потоков и инверсии зависимостей заслуживают больше внимания. Хотелось бы больше альтернатив mobx и angular2. Я, вышеозвученные характеристики реализовываю в reactive-di. А всем желаю уделять больше внимания базовым, идейным вещам, а меньше маркетинговым - хайповым, тогда наша работа станет комфортнее.

- github.com/zerkalica/reactive-di
- medium.com/@sergey_yuferev
- nexor@ya.ru

Юферев Сергей, qiwi.ru