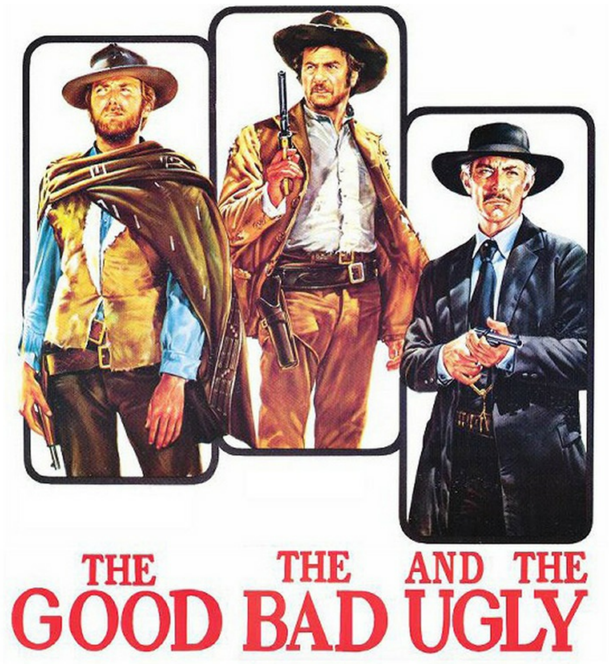
Как прекратить войну фреймворков

Меня зовут Сергей. Я работаю фронтенд-разработчиком в компании QIWI. У нас много проектов, в которых используется react, redux, mobx. В данный момент я, вместе с небольшой командой, работаем над проектом, где кроме react используется reactive-di. Это самописная библиотека для управления состоянием, построенная на принципах инверсии зависимостей. Мой доклад будет не столько про нее, сколько про причины, приведшие к написанию reactive-di. И вот они...

Убийцы архитектуры

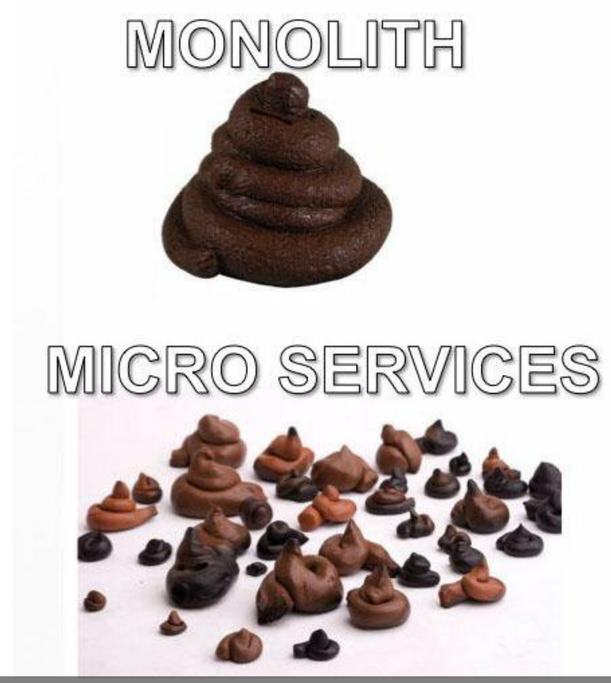


У Petka Antonov, создателя bluebird промисов, есть статья "Убийцы оптимизации" (Optimization killers). Где говорится каких конструкций в коде следует избегать, что бы код не стал менее производительным. Аналогично, я попробую рассказать об

конструкций в коде следует избегать, что бы код не стал менее производительным. Аналогично, я попробую рассказать об "убийцах архитектуры", т.к. коде, который затрудняет поддержку программ, навигацию по ее модулям, ведет к появлению багов в run time, а не в compile time.

А что не так?

Если не брать в расчет хайп, то как узнать, хороший или плохой фреймворк или библиотека, которую вы собираетесь использовать. Кто делал smart компонент, а потом переписывал на smart+dumb? Кто огребал с невозможностью переиспользовать или переопределить логику внутри компонента? Кто пробовал фетчить данные в componentDidMount и сталкивался с тем, что разные компоненты запрашивают одни и те же данные. Кто сталкивался с проблемами оптимизации и вколачивал в компоненты shouldComponentUpdate, а потом ловил с них баги. Как показать статус загрузки состояния (крутилку), да не всего целиком, а конкретного стора или модели. Немного про архитектуру...



Общая тенденция в програмостроении такова - одну большую проблему разбиваем на много маленьких. Как-то так, причем кучи

Общая тенденция в программостроении такова - одну большую проблему разбиваем на много маленьких. Как-то так, при этом кучи еще разноцветные. Большинство js-фреймворков - это обычно одно ядро, вокруг которого накручено много всего: react занимается и ui и состояние там есть и логика вокруг него. Однако, если посмотреть, как развивались другие языки, например, java, php, то там тоже были подобные решения, однако в итоге все придит к множеству мелких библиотек не связанных между собой или связанных через интерфейсы. Библиотеки легко собираются в фреймворк посредством некого клея: dependency injection. Даже говорят мета-фреймворк или (BUILD YOUR OWN) BYO.

4

Слои

Это слои - данные, представления, логика. Деление такое не с проста. 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 связаным с остальными частями.

5

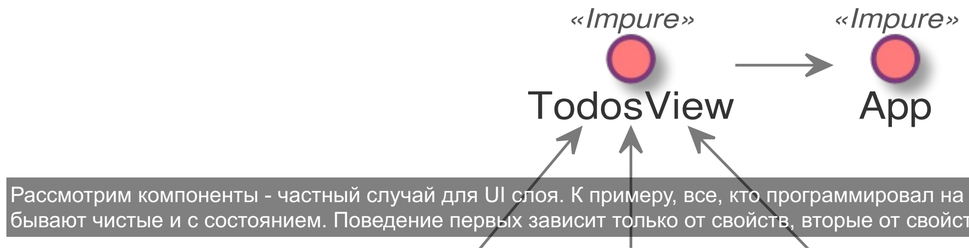
Связи

Я упомянул связи. Они бывают сильные: когда extends React.Component или React.createElement замаскированный под jsx. Так и слабые: интерфейсы на props, использование React.context вместо import. Для масштабирования и поддержки большого приложения важны как раз слабые связи. Однако сильные связи упрощают навигацию по проекту и отладку в отсутствии мощных IDE и др. инструментов разработчика.

Кирпичи

Есть еще базовые кирпичи - это функции, которые мы рассовываем по слоям и связываем между собой. Сами функции бывают с контекстом или чистые. Классы можно свести к функциям, контекст которых это this.

Чистые функции и состояние



Рассмотрим компоненты - частный случай для UI слоя. К примеру, все, кто программировал на реакте, знают, что компоненты бывают чистые и с состоянием. Поведение первых зависит только от свойств, вторые от свойств и еще от состояния.

Чистый компонент

- *dumb, presentational*
- view = component(props)
- Легкость переиспользуемости
- Рефакторинг: O(depth * props)

Для чего нужен компонент, результат которого - 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.

Чистый компонент

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

Первый киллер. Реализация чистого компонета React. Вроде никаких связей с другими частями приложения. 10

На самом деле нет

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

Но если собрать с babel-preset-react то появится прямая зависимость от React, если использовать другие решения с jsx: deku, inferno, vue, то суть таже. Нельзя переиспользовать чистый компонент в другом фреймворке, поддерживающим JSX. Поэтому мы имеем кучу реализаций несчастного bootstrap с material на разных фреймворках именно из-за этой сильной связи, кроме желания выпендриться, изобретая 15й стандарт. Кроме JSX нужны стандарты на то, во что он компилируется. 11

По-настоящему чистый

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

```
}
```

Например, ослабить связь можно, например добавив в конец аргумент, реализующий интефейс createElement. Такой компонент можно где угодно переиспользовать, задав соотвествующий h. Я уже говорил о сложности рефакторинга O(depth * props), из-за него в большом приложении не бывает только чистых компонент.

Компоненты с состоянием

- *smart, hiorder, container*
- view = component(props, *state*)
- State - труднее переиспользовать
- Легче рефакторить - O((depth * props) + state)

Есть еще компоненты с состоянием. Из-за state компонент 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 сложности перетекает в state.


```
class CounterView extends React.Component<{name: string}> {
  state = {count: 1}
  constructor(props: Props) { super(props) }

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

```
        () => this.setState(({count}) => {
          count: count + 1
        })

      }>Add</button>
    </div>
  }
}
```

И вот еще один архитектурный убийца - это React-реализация такого компонента. Ограничений тут гораздо больше, чем сильная связь на React.createElement. Некоторые из них: React.Component - прямая завязка на реакт, без которой, к нему гвоздями прибитые flow и typescript не помогут выявить ошибки в props. Конструктор подчиняется неким негласным соглашением, что первый аргумент только props. setState - привязка к способу реакта менять состояние. Такой компонент нельзя использовать где-то вне реакта. Ключевой момент, тут нарушен принцип изоляции слоев - в UI присутствует и логика и верстка и состояние и нет способа изолировать одно от другого, без наворотов над ними, вроде react templates. Создали большую сложность, а потом решают архитектурные и проблемы оптимизации через усложнение приложения.

Статические импорты

```
import counterService from './counterService'

class CounterView extends React.Component {

// ...
          count: counterService.add(this.state.count)
// ...
}
```

Мы прям в компоненте писали count + 1, а что если надо вынести логику изменения состояния. Можно все фигачить на синглтонах и импортах. Осюда еще один киллер - связь с counterService жесткая, реализацию нельзя поменять на что-то с таким же интерфейсом. 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)
  // ...
}
```

С этим можно бороться через React.context, правда он страшный, фейсбуковцы 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 не очень хороший признак, если ошибки, связанные с типами приходится отлавливать в run time.


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

function CounterView(
  { name }: { name: string },
  { counter }: { counter: CounterModel },
  h
) {
  return <div>
    {name}: {count.counter}
    <button onClick={
      () => counter.count = counter.counter + 1
    }>Add</button>
  </div>
}
```

Можно придумать компонент с более четкими границами слоев. Совместимый с flow и nuclide ide. 1й аргумент - props, 2й - контекст, 3й - createElement. Пример полностью абстрагирован от ui-фреймворков, от state-management фреймворков. Среда исполнения компонента обеспечивает связь CounterModel и CounterView, реактивность counter.counter + 1 и адаптацию к react, mobx, angular, да чему угодно, только адаптеры нужно написать.

Я тут все про реакт, да про реакт, а что с другими... Vue, aurelia, glimmer, deku, yo-yo, cyclejs, funnel... Вы поняли да? Стоит пожалуй только angular2 упомянуть, т.к. среди всего это камин аута со слоями, связями и типизацией, трюкам со стримами, только он чуть приподнялся на ступеньку.

```
@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)
  }
}
```

Ребята из команды angular2, не стыдятся контекста, а гордо его показывают в конструкторе. Вау, пропсов в нем нет. Они декоратором @Input помечены. В итоге это гораздо ближе к нативному синтаксису typescript. Правда без типов это все не очень удобно использовать. 20

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

Что бы магия заработала, ангуларовцы слегка прогнув микрософт с их тайпскриптом, записывают ссылку на CounterService в функцию Counter. Такая наколеночная страшненькая рефлексия, 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
  }
}
```

Однако и тут есть ложка дегдя и не одна, например CounterView - это реактивная модель со свойством counter и сервис с addCounter, прибито это все через декоратор @Component к шаблону counter.component.html. Один к одному сцепили шаблон, описание контракта к этому шаблону, модель, и логику по работе с ней. Поимев проблемы с выведением типов в html-шаблоне, если JSX flow понимает, то тут только за счет сторонних решений, плагинов к IDE и т.д, которые заведомо в этой области будут хуже работать, чем typescript, т.к. он специализированный инструмент.

redux

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

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

// 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
```

в react-redux (как и в mobx-react) React.context пытаются упрятать в Provider. И тут несколько проблем. 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. в combineReducers я ошибся и написал xyz вместо user, а в mapStateToProps state.user будет undefined и flowtype не отловит это. В третьих, никак не проверяется интерфейс того, что мы подали в Provider. В четвертых, в контекст через провайдер нельзя передать сервис, только стор. В пятых, задача контекста, которую решает Provider, не имеет отношения ни к react, ни к ui, она относится к способу связывания слоев в приложении, к внедрению зависимостей.


```
declare class Provider<S, A> extends React$Component<
  void,
  { store: Store<S, A>, children?: any },
  void
> { }
```

Однако Provider, прибит к React$Component

```
<Router history={history}>
  <Route path="/" component={App}>
    <Route path="foo" component={Foo}/>
    <Route path="bar" component={Bar}/>
  </Route>
</Router>
```

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

```
function CaseComponent({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 прибивать роутинг к реакту, а потом делать убыстренный клон реакта inferno, и копипастить его туда, также как inferno-mobx, inferno-redux, и еще несколько. 27

Прибитость React:

```
this.setState({counter: 123})
```

Избыточность Redux:

```
dispatch({type: 'INCREMENT', val: 123})
```

Магия Angular2:

```
this.heroes = heroes
```

Магия Mobx:

```
store.counter = 123
```

Каждый предлагает свой способ менять состояние. setState - прибит к реакту, что бы изменить состояние в редукс, компонент должен знать о интерфейсе dispatch, знать о интерфейсе экшена, либо писать дополнительный код для адаптации к redux-thunk. В ангуларе вся магия работает почему-то только в компоненте. mobx самый оптимальный вариант. К тому же, это самый простой интерфейс по чтению/записи состояния. Ключевой момент, он не предполагает завязанности на какую-либо библиотеку. 28

Оптимизация

Оптимизации - код, который не имеет отношения

к результату задачи, но содержит баги.

Если речь заходит об оптимизации в коде приложениея, то смело выкидывайте фреймворк на котором вы пишите. Я хорошо помню, как с появлением react было много хайпа о том, что VDOM, лежащий в его основе, настройко быстрый, что делает оптизизацию за вас. По сути - это кэш перед выводом в браузер. Тем не менее вам дается фича (а точнее костыль) вроде shouldComponentUpdate, которая позволяет делать тонкую оптимизацию (точнее неявно создавать баги и плодить инфрастуктурный код)

React

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

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

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

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

Код, отвечающий за оптимизацию в вашем приложении, замечательный. Он помогает вам скоротать немало часов, отлавливая баги на продакшене и flow с ts тут бессильны. А код с forceUpdate считается не нормальный в react. А вы заметили тут баг?

Angular

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

```
class CounterView {
  counter = 0
  constructor(private cd: ChangeDetectorRef) {}

  ngOnInit() {
    this.counter = this.counter + 1; // application state changed
    this.cd.markForCheck(),
  }
}
```

Но почему-то такой код считается нормальным в angular2. По-умолчанию, ангулар считает, что любое свойство в компоненте - Observable и оборачивает его, так что присвоение this.counter, вызывает перерендеринг. Это тормозной на больших приложениях механизм, который правильнее было бы не делать в ангуларе вовсе, а вынести в стороннее решение, вроде mobx.

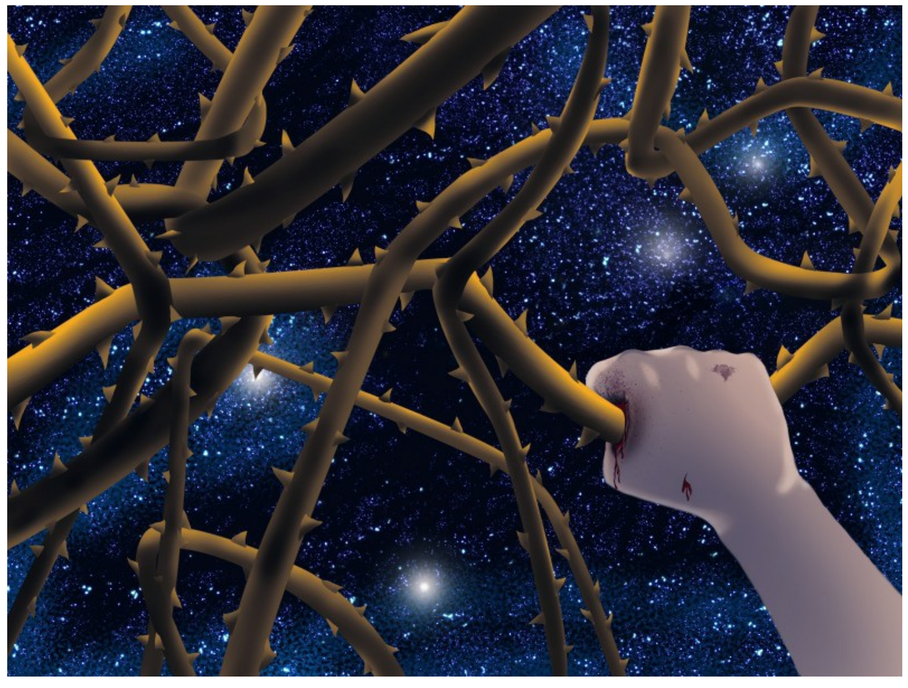
Mobx

```
function CounterView({count, name}: {count: number; name: string}) {
  return <div>{name}: {count}</div>
}

const CounterContainer =  observer(CounterView)

<CounterContainer />
```

Про оптимизацию в mobx-подобных решениях. Тут все гораздо лучше, т.к. оптимизация происходит раньше, в слое данных, а не в VDOM (react) или в компонентах (angular)

Плохо

- Оптимизации в коде приложения
- Смешение слоев setState, component+model+service
- Выдумывание базовых вещей: React.createClass, new Vue({data, watch, computed, methods})
- Завязка на библиотеки: React.createElement
- Маркетинговые бенчмарки



Путь тернист и конца пока не видно

Что стоит пытаться делать?

- Каркас для связей слоев data - ui - code (DI)
- Изоляция: интерфейсы
- Интеграция: ng-modules, ide, flow, ts
- Модульное безопасное состояние (mobx)
- Оптимизацию задвинуть за сцену
- Упростить рефакторинг O(depth * props + state)
- Оставаться KISS, BYO

Это критерии для тех, кто хочет объективно оценивать фреймворки, а не просто вестись на хайп. Экспериментаторам, которые пытаются писать свои, привнося что-то новое. Вся свистопляска фремворков вокруг компонента - это как покрасивше оформить, да и что б всякие flow с typescript atom-ами работали, копипасты небыло. 35

reactive-di

В начале я упоминал reactive-di. Коротко расскажу про эту библиотеку. 36

Model

```
class Counter {
    count: number = 0

}
```

Вот, например, как выглядит модель. Чистый класс с дефолтными свойствами, без методов. Легко сериализуется, всегда есть значение по-умолчанию, что бы избежать undefined багов.

View

```
function CounterView(
    {name}: { name: string; },

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

Чистая верстка, с контрактом и разделением на публичный интрефейс (props) и приватный (context). createElement, скрыт и подставляется babel-плагином.

Action

```
@actions class CounterActions {
  _counter: Counter

  constructor(counter: Counter) {
    this._counter = counter
  }

  add() {
    src(this._counter).set({
      count: this._counter.count++
    })
  }
}
```

Сервис, которые предоставляют методы, по сути экшены, меняющие состояние: тут может быть валидацая, fetch на сервер и т.д.

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)
    })
  }
}
```

Логика актуализации состояния Counter задается в специальных сервисах. Rогда первый раз отрендерился хотя бы один компонент использующий модель Counter, выполнится pull и состояние Observable с этого момента будет управлять Counter ом.

- github.com/zerkalica/reactive-di