ОТВЕТЫ НА ПЕРВОЕ ДЕМО

EVGENY GUSEV

1) КАК СДЕЛАТЬ SOLID - НУ ЕСТЬ ЖЕ ТЕПЕРЬ ES6, TS, DART

2) НО ВЕДЬ ЕСЛИ ВЗЯТЬ ДРУГОЙ ЯЗЫК - ТО ВОТ ТЕБЕ СОЛИД, И БЕЗ ФРЕЙМВОРКОВ

Сделать SOLID может быть много способов, важна цена реализации. Например, насколько много ручной работы по достижению SOLID автоматизируется фреймворком (DI), сохранится ли безопасность типов после этой автоматизации, насколько удобной будет навигация по проекту (контракты скрывают

реализацию и затрудняют ее поиск по интерфейсу)

4) КОМПОНЕНТ - ЭТО КОМПОНЕНТ, ЕДИНИЦА КАКОЙ-ТО СИСТЕМЫ

Под компонентом во фронтенде большинство понимает все же что-то похожее на компонент реакта или ангулара. Важно разделение ответственности, например по принципам MVC. Смысл в упрощении рефакторинга, когда меняя один слой, мы не трогаем другой.

Рассмотрим к примеру реакт. Мы не можем заменить реактовый vdom на другой или вообще использовать реальный DOM, мы пока еще не можем выкинуть код, обслуживающий setState, React.context или React.propTypes и заменить на свой. Мы не можем использовать чистый компонент вне React из-за наличия React.createElement, который является сайд-эффектом.

В компонентах реакта и им подобным плохо с разделением ответственности, т.к. внутри она состоит из мешанины этих слоев. render

содержит верстку, onComponentDidMount провоцирует на написание логики в компонентах, setState прибивает данные и способ работы с ними к компонентам, React.PropTypes костыльная замена flow или ts, работающая только в runtime.

В ангулар у компонент тоже слишком много ответственности: состояние, логика, livecycle, еще она распределена странно:

```
@Component({
  selector: 'my-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: [ './heroes.component.css' ]
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  getHeroes(): void {
    this.heroService
        .getHeroes()
        .then(heroes => this.heroes = heroes);
  }

  add(name: string): void {
```

HeroesComponent - это такая моделька, с сеттерами, геттерами, lifecycle обработчиками и внутренним состоянием, внутренней логикой работы с этим состоянием, только эта моделька еще прибита гвоздями к html и css. Хотя логичнее было бы наоборот, view делать зависимым от модели.

Например в php есть Doctrine ORM, где отдельно

тонкая (anemic) модель, отдельно lifecycle-callbacks, сервисы для работы с ними, отдельно view.

На мой взгляд было бы правильнее с точки зрения разделения ответственности все-таки признаться, что шаблоны - чистые функции, с сигнатурой и типами, на основе которых можно строить метаданные и зависимости.

Далее пример компонента Hello с соблюдением separation concern.

```
@lc(UserHooks)
class User {
    name = ''
}

class UserHooks {
  onFirstAccess(user: User) {
    fetch('/user').then(data => {
      user.name = data.name
    })
  }
}

class UserErrors {
  name = ''
```

5) SOLID - ТЕБЯ СПРОСЯТ, А НАФИГА ОН НУЖЕН? НЕ ВСЕ ПРИШЛИ С БЕКЕНДА

Лучше мне наверное не говорить про SOLID. Если просто озвучить принципы, мало кто поймет, а с

примерами это слишком большая тема. Если вкратце - набор практик, облегчающих поддержку большого приложения. Например, как заменить TodoItemView на MyTodoItemView, если первый в сторонней библиотеке и ее код трогать нельзя или как заменить React на virtualDOM или bel или что-то еще без пересборки компонент.

6) ПРО АНГУЛЯР - МОЖНО И БЕЗ DI, В ОТЛИЧИЕ ОТ ПЕРВОГО

Можно, любая хорошая библиотека di-friendly. Вопрос какой ценой, DI автоматизирует связывание кода, уменьшая кол-во ручного труда и ошибок, связанного с ним, ценой знания простых принципов по организации приложения. В эта цена приемлема для сложных приложений и средних.

7) НЕНАВЯЗЧИВЫЕ ПОТОКИ - НЕ СЛИШКОМ ЛИ МНОГО МАГИИ. И В ЧЁМ МИНУСЫ?

Что такое магия? Многие знают как внутри

работает реакторвый VDOM, а setState, а React.context? Или @input/@output в ангуларе. Или какой маг делает ререндер вьюшки в ангуларе в `this.heroes = heroes`?

```
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(
    private heroService: HeroService,
    private router: Router) { }

  getHeroes(): void {
    this.heroService
        .getHeroes()
        .then(heroes => this.heroes = heroes);
  }
```

Примеров можно привести много. Лучше оперировать более объективными понятиями, например: связанность, цельность, кол-во строк копипаста на новый код, простоту абстракции, похожесть абстракции и поведения работы с ней на какие-то привычные вещи, нужность этой абстракции в решаемых задачах.

Ненавязчивые потоки это про рендер при `this.heroes = heroes`, только отвязанный от компонент, ангулара, zones и кучи другого барахла.

Как у любого решения есть плюсы и минусы. Плюсы - это похоже на что-то очень простое и

понятное, не требующее изучения сотни методов апи.

Минусы - некое условие, что все данные - обертки и подписывание происходит автоматически при чтении данных, не всегда там, где нужно, как в mobx. Можно отметить алгоритмическую сложность этих решений (это не redux, наивная версия которого пишется за час): нетривиальные приемы по оптимальному обходу дерева зависимостей, трассировке данных, дедупликации, кэшированию и т.д. Как следствие багов больше, точнее они перераспределены из прикладного слоя в инфраструктурный, но это скорее фича, тот же redux наоборот - простота ядра за счет усложнения перифирии, что в целом хуже для экосистемы.

Но такие решения все-равно победят из-за своей видимой простоты, как победил react и mobx. Тут можно провести аналогию с ORM в java/c#/php: они дают профит в 80% случаев на больших архитектурах, однако в 20%, когда много сложных выборок, группировок и т.д., лучше сделать прямой запрос к базе, т.е. просто не используем эту технологию, там, где не надо.

9) LIFECYCLE НА МОДЕЛИ - МНОГО МАГИИ И ПОТОМ ФИГ ПОДДЕРЖИШЬ. МАГИЯ!

Про магию я что-то уже говорил, отвечая на п. 4. Кратко, магия - это здорово, это хорошо, если это белая магия.

Почему lifecycle в компонентах не магия, а в моделях - магия, почему, `this.heroes = heroes` в компонентах ангулара не магия, а в mobx - магия?

Вся суть в разделении ответственности, lifecycle - это про данные, это не про фронтенд вообще, а во всех популярных фреймворках упорно прибивают их именно к фронтенд-компонентам. Компонент - это просто потребитель данных: отрендерился - завладел деревом, дерево актуализировалось, удалился из DOM - перестал владеть деревом, дерево заснуло.

Актуализация данных должна быть ближе к данным, а не к view. Идее lifecycle в моделях больше лет, чем в компонентах, вроде lifecycle в doctrine. Если очень надо, могу аналогию

поискать в java и c#.

10) ПОЧЕМУ ЭТИ РЕШЕНИЯ ХОРОШИ, В ЧЁМ СОЛЬ? ЧЕМ ХОРОШИ MOBX CELLX. ТЫ ПОДРАЗУМЕВАЕШЬ, ЧТО АУДИТОРИЯ ЭТО ЗНАЕТ.

Почему хороша ненавязчивая реактивность я уже немного объяснил в п.7: простым поведением, минималистичным апи.

Почему хороши хуки в моделях. Без них мы пишем такой код:

```
componentDidMount() {
  this.state.todos = service.fetchTodos()
}
```

Является нарушением инкапсуляции, особенно это заметно, когда несколько компонент используют одни данные, часть ответственности по актуализации состояния делегируется компоненту и этого никак не избежать. Актуализация состояния - ответственность слоя данных и должна оставаться в нем (пример с HelloView в начале).

```
class Todo {}
```

```
class Todo {}
const todos: Todo[] = []

const service = {
  fetchTodos(): Todos[] {
    fetch('/todos').then(someMagic)
  }
}

class MainTodosView extends React.Compoent
  state = {todos: []}
  componentDidMount() {
    this.state.todos = service.fetchTodos()
  }
  render() {
```

Про cellx. Есть класс задач, который удобно решать именно на lifecyle в моделях, например, синхронизация значения с асинхронным хранилищем

Можно много говорить о плюсах и минусах, магии и т.д., но эти идеи интересно развивать просто как альтернативу потокам и сервисам, уменьшать кол-во магии, делать прозрачными и т.д., смысл поиска - доказать или опровергнуть состоятельность этой идеи.

11) REACTIVE-DI - НИКАКОГО ВСТУПЛЕНИЯ. ЛЮДИ НЕ ЗНАЮТ, ЧТО ЭТО ТАКОЕ

Могу что-то очень краткое, звучащее вкусно, но не более. Я свой доклад планировал про проблемы экосистемы js, про войну

фреймворков, reactive-di это большая тема. Это как если бы я попытался за один доклад про ангулар рассказать вместе с принципами SOLID и примерами.

12) ЛНПЧ И ПРОЧЕЕ. ЗВУЧИТ КАК УТОПИЯ. ДЕЛАЙТЕ ХОРОШО - БУДЕТ ХОРОШО

Я пытаюсь вывести объективые параметры оценки качества кода, на мой взгляд, SOLID, cohesion, coupling не достаточно и долго объяснять это фронтенд-разработчикам.

Легко написал, легко прочитал, легко поменял, в ногу не выстрелил - понятно что все они не достижимы, имеет смысл поиск баланса. И я не просто их озвучил, я еще немного раскрыл суть каждого.

Легко написал - минимум шаблонов, конструкции языка вместо конструкций библиотек (class vs React.createClass), автоматизации рутины: подписывание/отписывание (mobx), инжекция зависимостей (inversify, angular2).

Легко прочитать - контракты как первичная

документация, фрактальная масштабируемость: когда модуль по структуре похож на все приложение. Тут много работы, много ли примеров по интеграции модулей в приложение, примеров разделения их по слоям и т.д., я только про angular2 с их ng-modules могу вспомнить.

Легко поменять - модули содержат набор слоев MVC с четкими границами, контракты, слабая связанность, инструменты для анализа, т.к. слабая связанность затрудняет навигацию по проекту

В ногу не выстрелить - Безопасность кода на типах, максимальный отлов ошибок на этапе статического анализа (tcomb, React.propTypes - костыли из-за отставания flow и ts, в качестве правильного решения можно rust привести). В общем, выбор языка решает каким будет экосистема и сообщество программистов вокруг него, хреновый язык - хреновое и сообщество.

Все это можно выразить чуть иначе: набор инструментов контрактного программирования с низким порогом вхождения, ненавязчивая реактивность, безопасность типов

Многие из этих штук можно реализовать прямо

сейчас даже на js, инструменты есть, а некоторые додумать - например "Как просто реализовать связь интерфейсов и реализаций", так что не утопия.

13) В ПРОВАЙДЕРАХ В АНГУЛЯРЕ ЕСТЬ ПРОВЕРКИ. ИЛИ ЧТО ИМЕЛОСЬ В ВИДУ?

Я про статический анализ аргументов в декораторах. По примерам из angular di:

```
// An object in the shape of the logger service
let silentLogger = {
  logs: ['Silent logger says "Shhhhh!". Provided via "useValue"'],
  log: () => {}
};

[{ provide: Logger, useValue: silentLogger }]
```

Как в последней строчке, в статике, а не runtime осуществляется проверка, что silentLogger implements Logger?

Или вот:

```
[ NewLogger,
  // Not aliased! Creates two instances of `NewLogger`
  { provide: OldLogger, useClass: NewLogger}]
```

А если NewLogger не instanceof OldLogger ?

Или еще один пример

```
@Injectable()
class EvenBetterLogger extends Logger {}
```

```
[ UserService,
  { provide: Logger, useClass: EvenBetterLogger }]
```

Если забыл @Injectable(), то как об этом узнать на этапе сборки, а не выполнения.

Иными словами насколько хорошо сделана статическая проверка composition root?

В c# ninject, например сделано неплохо:

```
public class WarriorModule : NinjectModule
{
    public override void Load()
    {
        this.Bind<IWeapon>().To<Sword>();
    }
}
```

Тогда, как в его клоне на ts InversifyJS ужасно:

```
@injectable()
class Ninja implements Ninja {

    private _katana: Katana;
    private _shuriken: Shuriken;

    public constructor(
      @inject("Newable<Katana>") Katana: Newable<Katana>,
      @inject("Shuriken") shuriken: Shuriken
  ) {
        this._katana = new Katana();
        this._shuriken = shuriken;
    }

    public fight() { return this._katana.hit(); };
```

"Newable", "Shuriken" - источник багов. И это непроработанность спецификации interface as

value в runtime в ts и is.