

Bitmix CMS

Что это и зачем?

01

Бизнес-ценность

Бизнес имеет потребность находить новые каналы взаимодействия с клиентами, поскольку это позволяет наращивать их количество и [... всем и так всё понятно надеюсь ...].

Для реализации подобных потребностей рынок сегодня предлагает массу решений, начиная с кастомной веб-разработки на заказ, продолжая разработкой на основе готовых шаблонов (bitrix, wordpress, joomla и.т.д) и заканчивая no-code решениями типа tilda, wix и так далее.

Кастомная
разработка

Template-based

No-code solutions

Pros/Cons у текущих решений

Зачем что-то менять/дополнять?

Рассмотрим плюсы текущих подходов

Custom Development

- Широкая функциональность;
- Современные технологии, так как стек постоянно обновляется;
- Хорошая подготовка к SEO, безопасность (всё, что можно назвать качеством, кароче)

Template Based

- Скорость разработки (+/-);
- Эффективный контент-менеджмент;
- Широкая функциональность (+/-), за счет того, что пишутся кастомные блоки;
- Масштабируемость (+/-). (Найти и обучить персонал довольно просто)

Zero-code

- Скорость разработки;
- Масштабируемость (персонала очень много, легко учить)
- Клиенто-ориентированность (зачастую клиент непосредственно участвует в создании сайта, без посредников в виде менеджеров)

Pros/Cons у текущих решений

Минусы

Custom Development

- Дорого
- Долго
- Много bottle-neck factor
(Очень зависим от таланта программистов, отсутствия у них депрессии и шизофrenии, соответственно очень трудно масштабироваться)

Template Based

- Дорого (почти любой фреймворк который можно расширять под свои задачи стоит бабок, и не малых. На плечах клиента это ренты, а на плечах студии это лицензия и курсы, и всё это выливается в немалую цену разработки)
- Зачастую функциональность готовых решений не позволяет написать то что нужно, и приходится писать кастом (Возможно это даже плюс, так как клиент платит :))
- Когда -- говно, программисты часто кончают жизнь самоубийством

Zero-code

- Нельзя сделать ничего сложного (Любая задача включающая в себя какую-то логику, визуальную или бизнесовую, требует полностью кастомной разработки, которая в инструментах zero-code делается очень сложно и ограничено)
- Немного дорого (самое дешевое, но все же нужно платить ренту + в случае премиальных блоков)

Мы можем изменить эту схему

Как Взять плюсы без минусов?

Можно ли создать решение одновременно **дешевое, масштабируемое, безопасное и гибкое?**

Если бы мы смогли реализовать подобное, то могли бы выйти на рынок с конкурентным преимуществом, делая тоже самое, что и остальные, но намного дешевле, быстрее и в больших объемах, тем самым захватывая рынок и покупая яхты

Финансы

Из анализа остальных решений можно вычленить, что стоимость разработки растет прямо пропорционально кол-ву **кастомной разработки и лицензирования.**

Это главные враги с точки зрения финансовой эффективности, а значит и возможной стоимости для заказчика, а значит и главного с точки зрения предложения преимущества.

Введем меру эффективности

Псевдо-математически :)

Учитывая, что стоимость часа zero-block конфигуратора примерно 250 рублей, судя по hh, а стоимость часа талантливого веб-программиста, который сможет делать кастом 1700, то есть пропорция **15:100**, либо, в случае с джуном пропорция будет $(250/750)=1:3$

То есть, если заменить джуна на зерокодера, мы получим выгоду в 3 раза, а если талантливого сеньера, то в 6 раз.

В этом собственно суть плана -- максимизация зеро-кодеров в процессах, классически реализуемых программистами. Вопрос как это можно максимизировать?

Масштабирование и безопасность

Правильное масштабирование IT-решения, тема хоть и сложная, однако хорошая изученная и решаемая, так что оставим пока это за скобками, точно также как и безопасность айти продукта, никакого новаторства здесь не предвидится

А Вам с последним “Гибкость”

Есть некоторые вопросы

Как можно максимизировать
количество зеро-код разработчиков,
сохранив гибкость создания
проектов?

Гибкость

С этой точки зрения, у существующих zero-code решений всё очень плохо.

Почти невозможно сделать сложные интерфейсы с использованием визуального редактора, не прибегая к абстрактному мышлению, импортам и прочими атрибутами веб-программирования.

Та база которую предлагают сегодня, является просто “накидыванием” созданных разработчиками блоков на холст, и настройкой их через интерфейс, полностью захардкоженный разработчиком.

При таком подходе, для всего, что касается глубокой кастомизации и настройки логики является работой дорогого разработчика, тем более стиснутого в рамки фреймворка, а тискать разработчика дорого, потому что дорого, и больно, потому что он может не захотеть это терпеть.

Но есть идея!

Идея

Чтобы продемонстрировать идею,
нужно повысить уровень абстракции,
лучше всего это сделать на “реальном”
примере

Закройте глаза, представьте что вы программист....
Вытрите слёзы, продолжим

В качестве базы, представим, что
старший талантливый программист
заботливо создал для нас некоторые
компоненты ->



1C Data parser

Рыночек порешал и все используют 1С,
черт бы их побрал, поэтому наш
старший товарищ создал для нас это

Этот компонент делает запрос на наш
сервак в эндпоинт с интеграцией,
обрабатывает ошибку и передает
данные в чилдрена, а также описывает
типы входов (props) и выходов (children
args)

```
1 type Item = {
2   id: string;
3   name: string;
4   price: number;
5 };
6
7 type Catalog = {
8   items: Item[];
9   pages: number;
10};
11
12 type Props = {
13   tokenSecretName: string;
14   catalogName: string;
15   children: (props: { catalog: Catalog }) => React.ReactNode;
16 };
17
18 export const OdinAssParser = ({ tokenSecretName, catalogName, children }: Props) => {
19   const parseCatalog = async () => {
20     fetch('/odinAssCatalogIntegration/get', { body: { tokenSecretName, catalogName } });
21   };
22
23   const { data, isLoading, isError } = useQuery('odinAssCatalog', parseCatalog);
24
25   return (
26     <>
27       {isError && <div>Failed to load catalog, try again</div>}
28       {isLoading &&
29         children({
30           catalog: {
31             items: data.items,
32             pages: data.pages,
33           },
34         })
35       }
36     </>
37   );
38 }
```

Ещё компоненты

Карточка магазина, просто выводим Верстку, плюс принимает в чилдрена фумер

Кнопка

Грид, добавляем стили для создания сетки и рендерим чилдрена

```
1 type Props = {  
2   image: string;  
3   title: string;  
4   price: string;  
5   children: React.ReactNode;  
6 };  
7  
8 export const ShoppingCard = ({ id, image, title, price, children }: Props) => {  
9   return (  
10     <div className={s.card}>  
11       <img className={s.image} src={image} alt={title} />  
12       <div className={s.title}>{title}</div>  
13       <div className={s.price}>{price}</div>  
14       <div>{children}</div>  
15     </div>  
16   ).  
1 type Button = {  
2   text: string;  
3   onClick: () => void;  
4 };  
5  
6 export const Button = ({ text, onClick }: Button) => {  
7   return <button onClick={onClick}>{text}</button>;  
8 };  
9  
10 type Props = {  
11   columns: number;  
12   children: React.ReactNode;  
13 };  
14  
15 const Grid = ({ columns, children }: Props) => {  
16   return (  
17     <div style={{ display: 'grid', gridTemplateColumns: `repeat(${columns}, 1fr)` }}>  
18       {children}  
19     </div>  
20   );  
21 };  
22  
23 |
```

Вернемся на уровень джуна...

Что мы теперь должны делать?

Имплементацию!

Просто импортируя компонент,
мы видем то, что он передает внутрь, в результате своей работы,
магия тайпскрипта



```
1 import { OdinAssParser } from "./OdinAssParser";
2
3 export const Page = () => {
4   return (
5     <div>
6       <OdinAssParser>
7         {([])
8           => </OdinAssParser>)
9       </div>
10      );
11    }
```

A screenshot of a code editor showing a TypeScript file. The code imports an `OdinAssParser` component and defines a `Page` function that returns a `div` element containing an `OdinAssParser` component. A tooltip is displayed over the `catalog` prop of the `OdinAssParser` component, listing several type definitions:

- (property) catalog: Catalog
- Region End
- Region Start
- afterAll
- afterEach
- anonymousFunction

А также, TS с радостью дает подсказку о том, что можно передать в компонент

```
export const Page = () => {
  return (
    <div>
      <OdinAssParser>
        {({ }) => (
          <Grid columns={3}>
            <ShoppingCard>
              <Button />
            </ShoppingCard>
          </Grid>
        )}
      </OdinAssParser>
    
```

```
4 export const Page = () => {
5   return (
6     <div>
7       <OdinAssParser>
8         {(() => (
9           <Grid>
10          )}
11        </OdinAssParser>
12      </div>
13    )
14  )
15}
16
17
```

После импорта всех компонентов TS орет на нас, что мы не прокинули нужные данные.

Мы как хорошие джуны берем и направляем данные из ОДИН ЭССА, в нужные места

TS Заботливо подсказывает, что OdinAss не будет работать без токенов и прочего, указываем

```
export const Page = () => {
  return (
    <div>
      <OdinAssParser>
        {{catalog}} => [
          <Grid columns={3}>
            {catalog.items.map((item) => (
              <ShoppingCard image={item.image} price={item.price} title={item.name}>
                <Button text='Купить' onClick={...} />
              </ShoppingCard>
            ))}
          </Grid>
        ]
      </OdinAssParser>
    </div>
  );
};
```

```
export const Page = () => {
  return (
    <div>
      <OdinAssParser>
        {{catalog}} =>
          <Grid colum
            {catalog.
              <Shoppi
                <Butt
                  <Shopp
                    )}>
          </Grid>
        )
      </OdinAssParser>
```

(property) catalogName: string
Region E
Region S
afterE
afterEa
anonymousFunction
anonymousFunction

Что сейчас произошло?

1) Импорт

*2) Создание иерархии
компонентов*

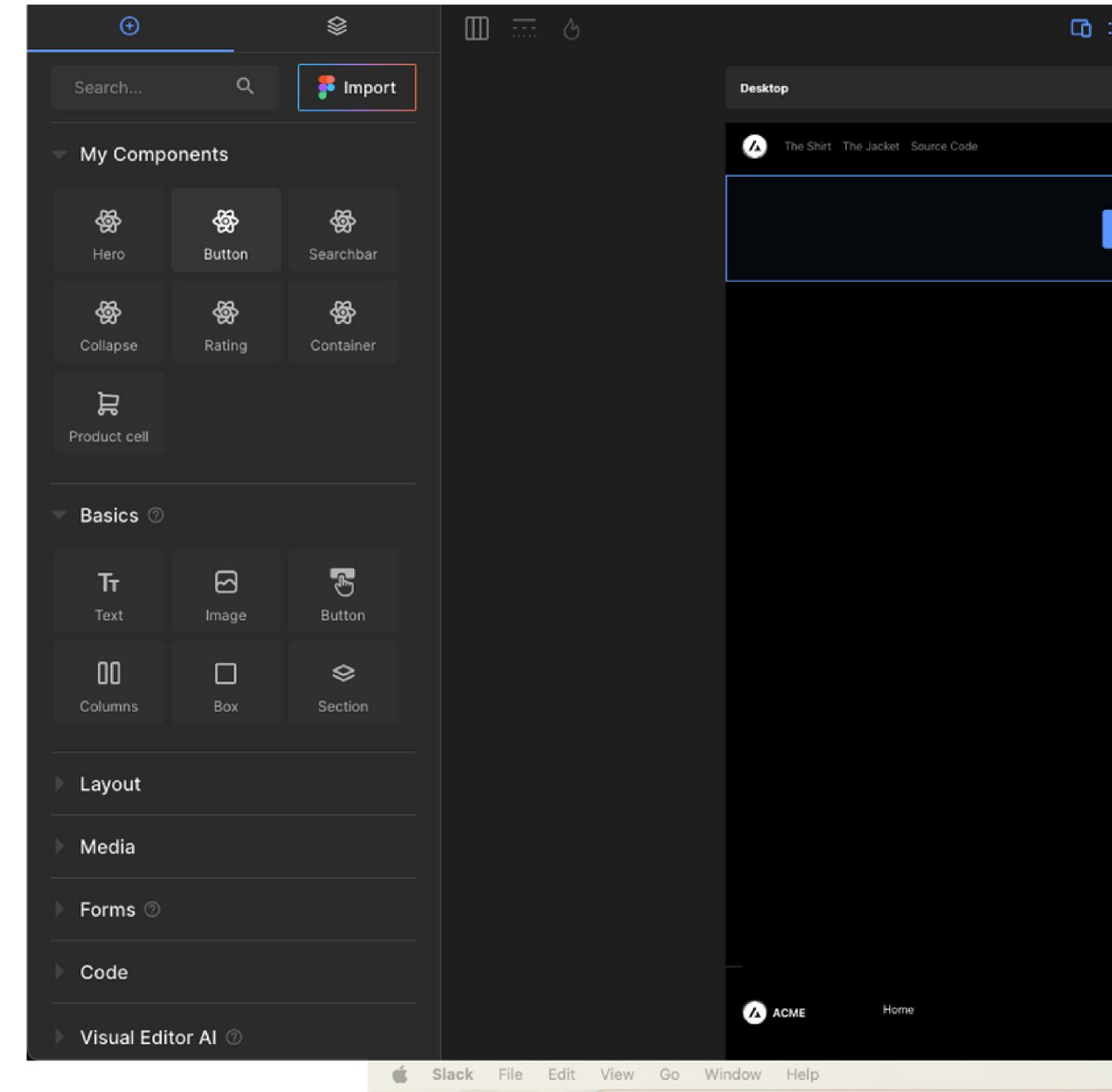
*3) Направление
данных*

**Из-за того, что typescript дал нам информацию о
данных -- мы смогли заняться просто их
менеджментом.**

**Эта задача легко решается с помощью Визуального
представления! И не требует особых знаний**

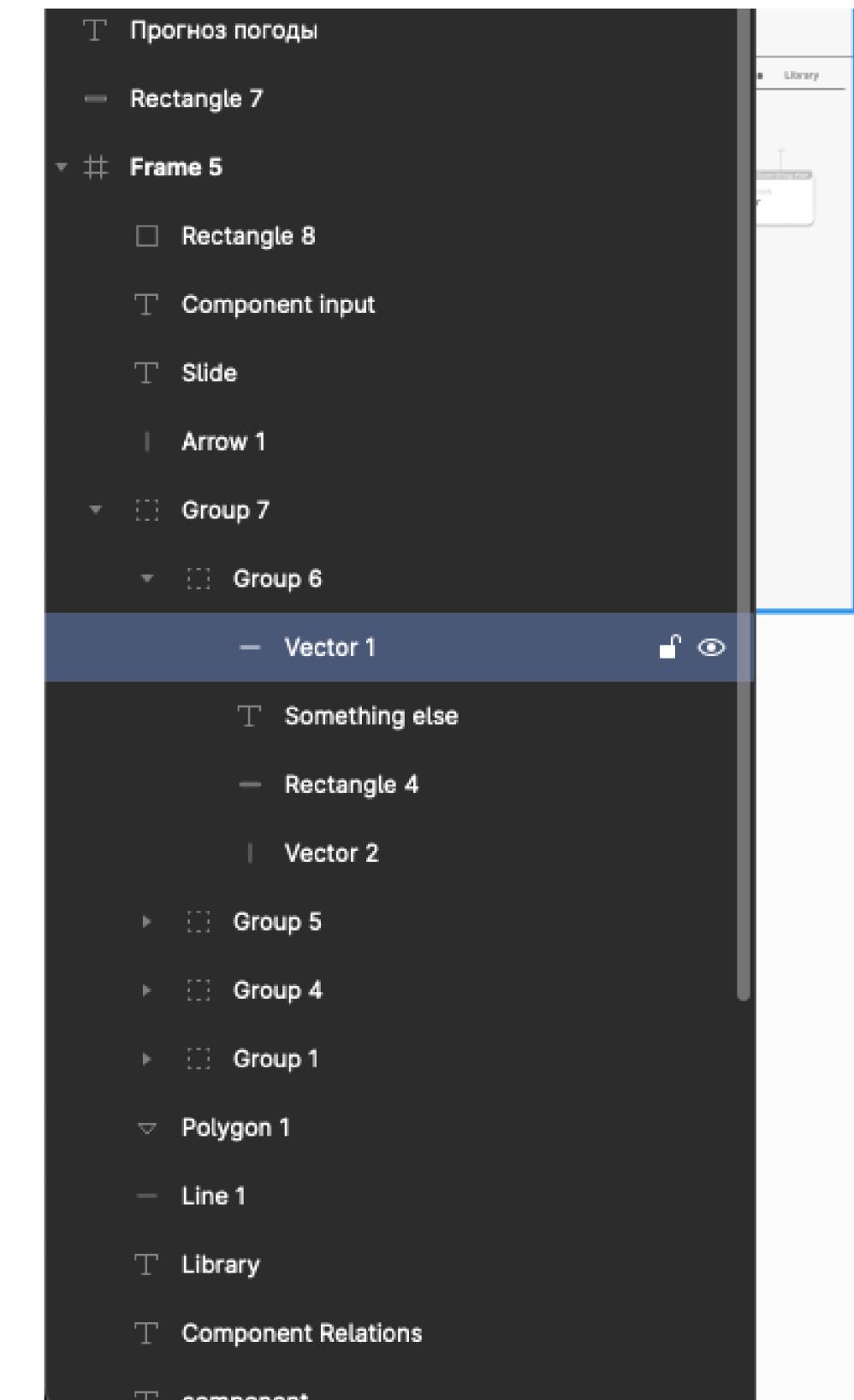
Как это может выглядеть в интерфейсе?

Задача импорта решена во всех известных CMS, это просто библиотека со списком доступных элементов и документацией по ним



Как это может выглядеть в интерфейсе?

Построение иерархии также является трибуальной задачей, так как направление вложенности строго вертикально исходящее



Направление данных

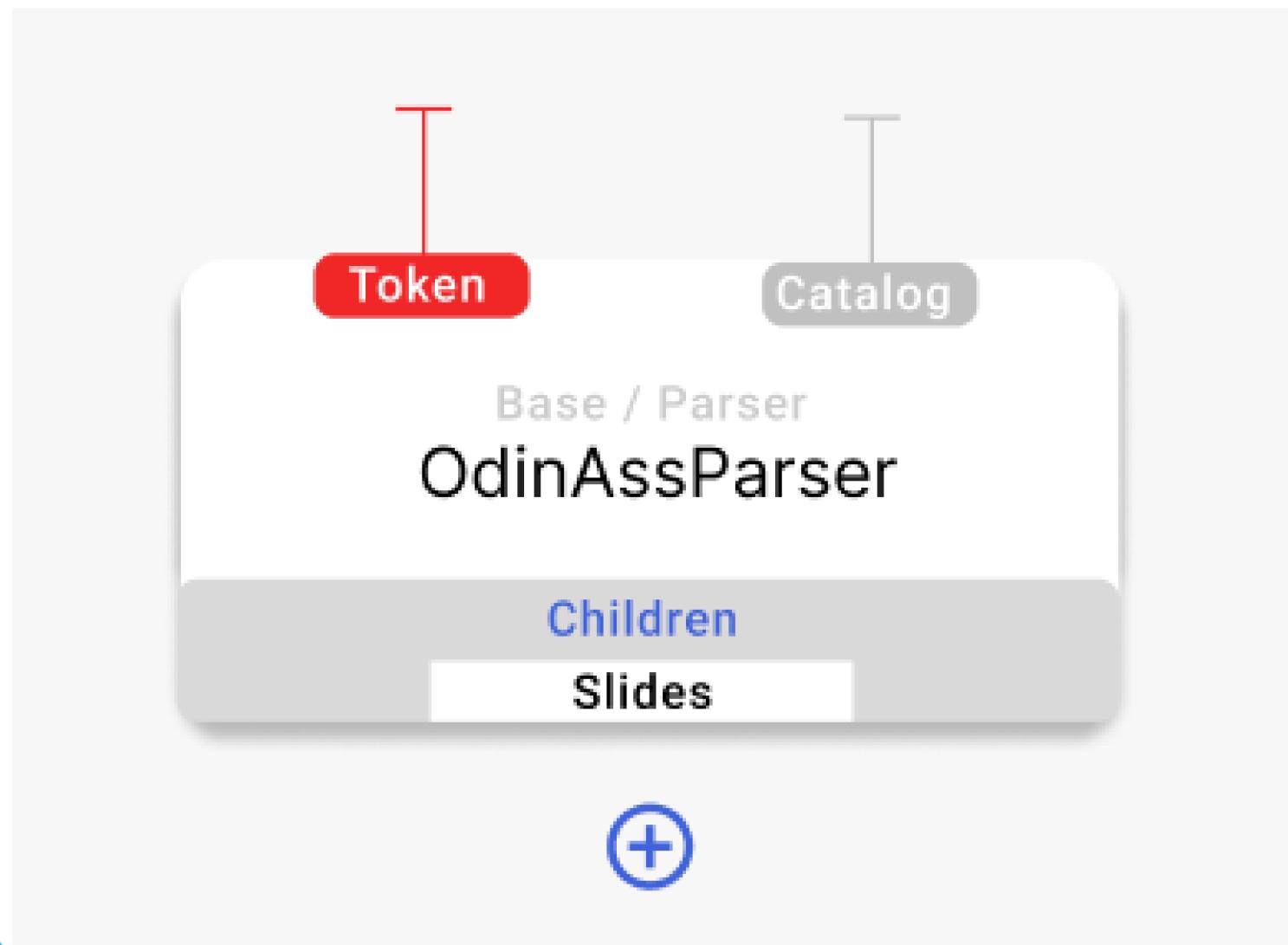
Единственная нетривиальная задача, которую нужно решить -- направление данных.

Есть много способов и не ясно, какой из них лучший

Направление данных

Мне кажется, мог бы сработать граф, тогда был бы такой процесс

После добавления компонента из библиотеки



- Сверху блока нарисованы входящие пропы
- Цветом обозначено их обязательность, допустим если проп указан красным, то это значит что он не **указан и обязательен** (как ошибка в IDE)
- В центре блока указана информация о блоке, название, категория И так далее , может ещё что-то о нем
- снизу указан слот для чилдрена, то, что он передает внутрь себя, при наведении можно посмотреть тип, которые формируется. Также тут работают дженерики для сложных компонентов
- и кнопка плюс,символизирующая добавление компонента в чилдрена

Направление данных



```
2
3  type Props<T extends object, K extends keyof T> = {
4    data: T[];
5    keyField: K;
6    children: (item: T) => React.ReactNode;
7  };
8
9  export const ObjectMap = <T extends object, K extends keyof T>({
10    data,
11    keyField,
12    children,
13  ): Props<T, K> => {
14    return (
15      <>
16        {data.map((item) => (
17          <React.Fragment key={item[keyField] as string}>{children(item)}</React.Fragment>
18        ))}
19      </>
20    );
21  };
22
23  const check = () => {
24    const array = [
25      { id: 1, name: 'example' },
26      { id: 2, name: 'example' },
27    ];
28
29    return (
30      <ObjectMap data={array} keyField="sdasdsda">
31        {
32          ♦+ (item) => [item.]
33          }
34        </ObjectMap>
35      );
36    };

```

Type '"sdasdsda"' is not assignable to type 'id' | 'name'. ts(2322)

Map.tsx(5, 3): The expected type comes from property 'keyField' which is implicitly 'IntrinsicAttributes & Props<{ id: number; name: string; }>, "id" | "name"

(property) keyField: "id" | "name"

[View Problem \(F8\)](#) [Quick Fix... \(⌘.\)](#)

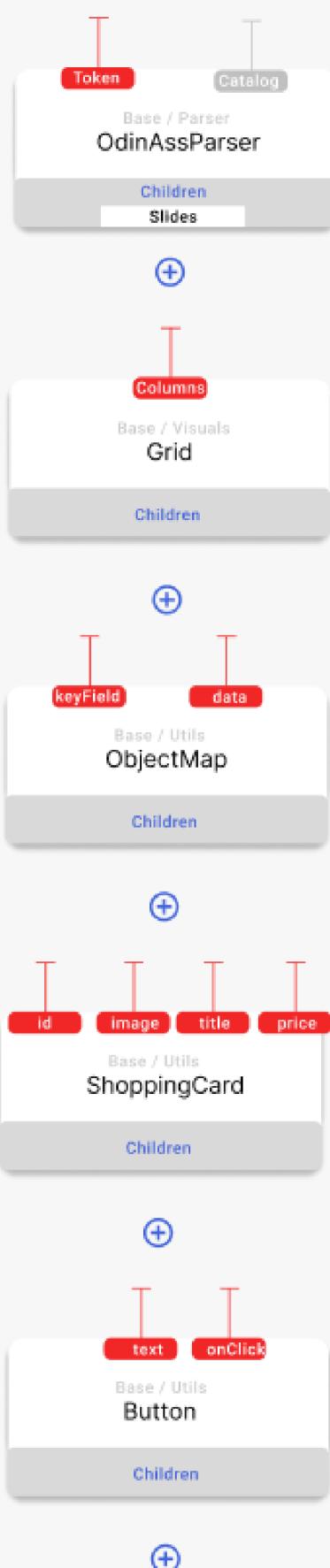
ray} keyField="sdasdsda">

id
name
(property) name: string

- Черта указывает на то что снизу живут дети
- Тут, так как нам нужно итерировать по слайдам, мы вынуждены добавить компонент ObjectMap, с такой реализацией, (как видно TS не даст совершить зерокодеру ошибку)
- Также автоматически считается дженерик того, что будет в чилдрене

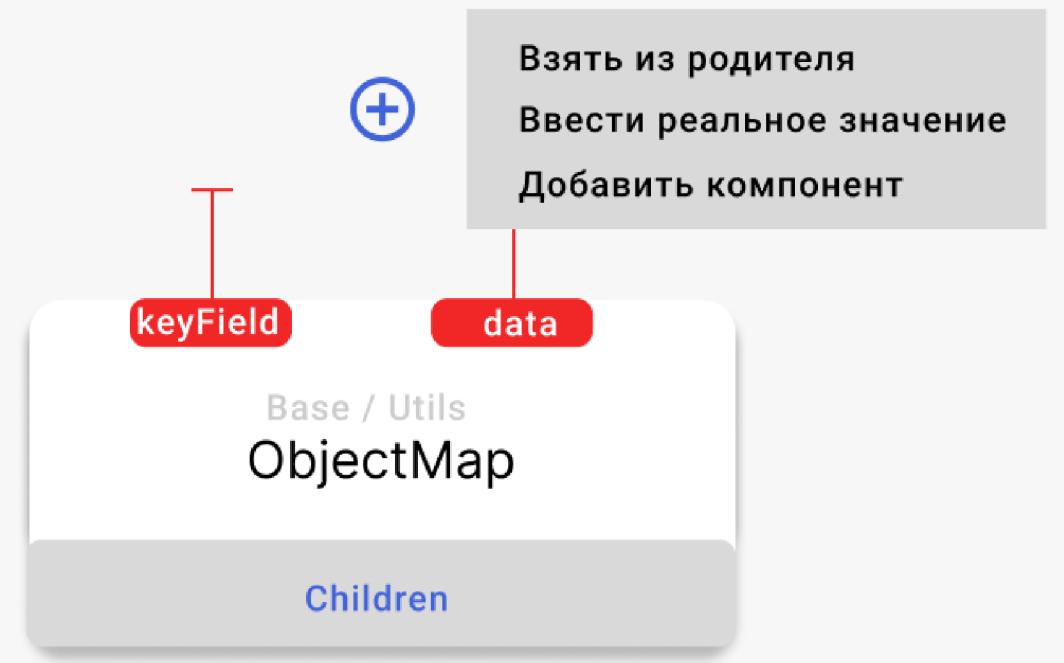
Направление данных

1

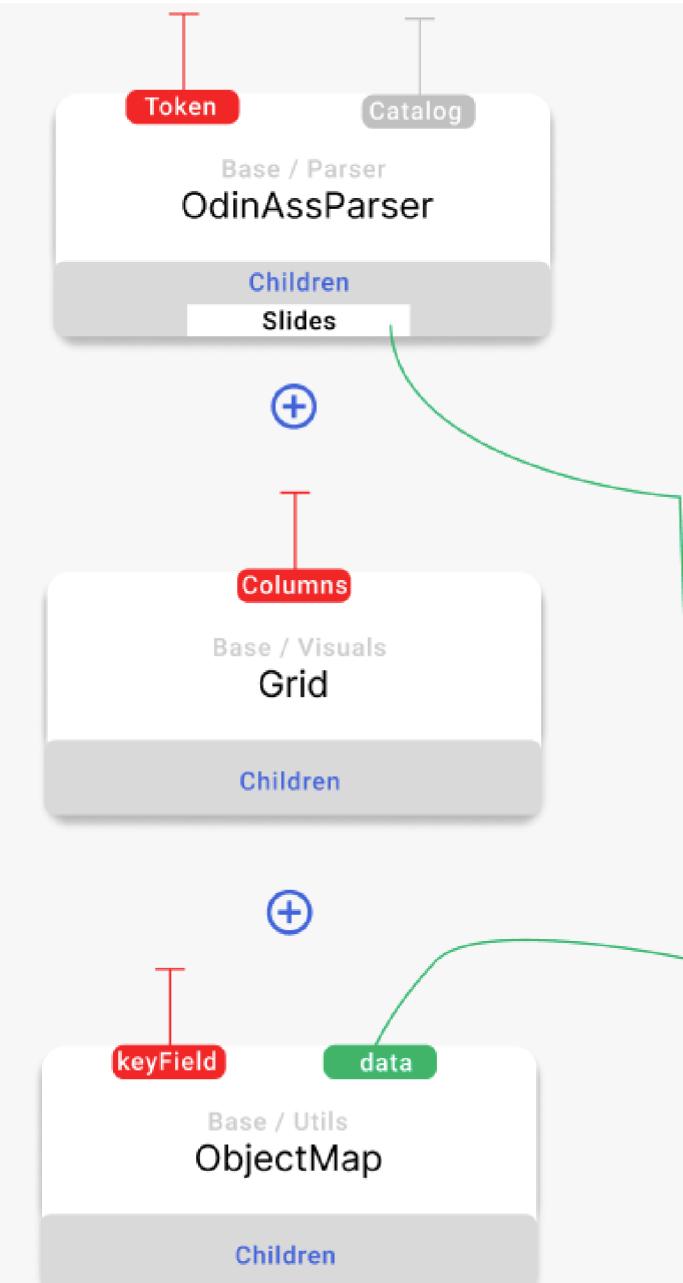


- После добавления всех чилдренов и создания структуры, зерокодер приступает к направлению данных по документации
- По клику на красные items появляется менюшка

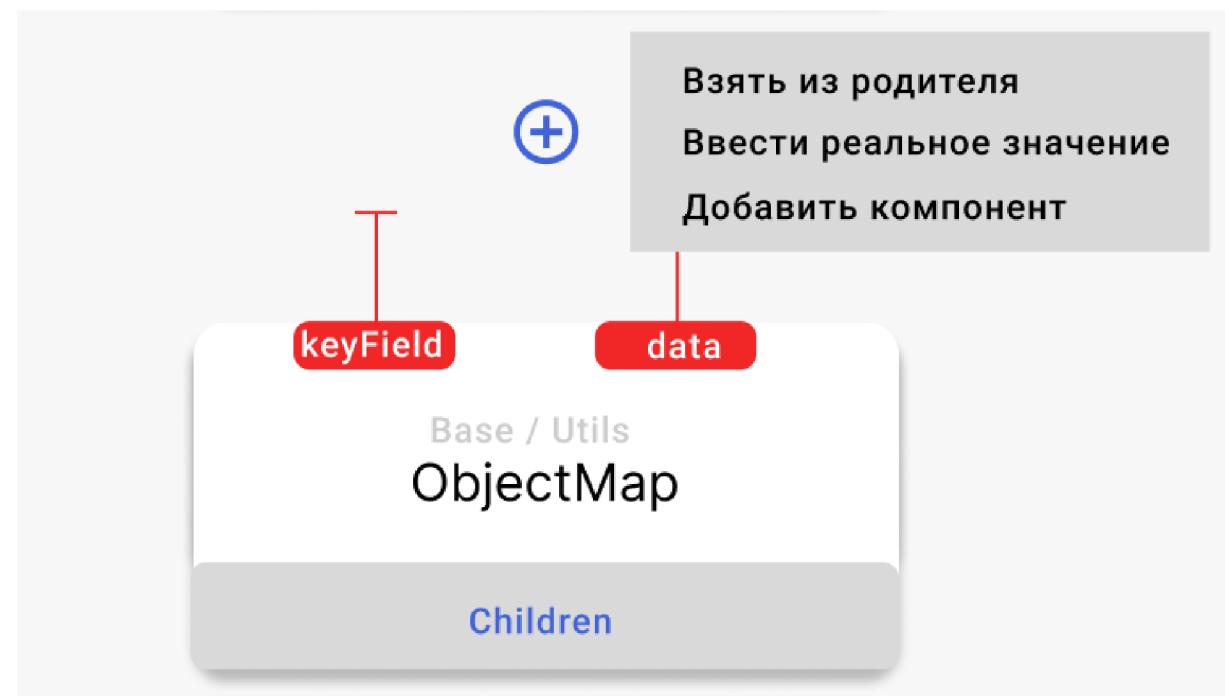
2



3

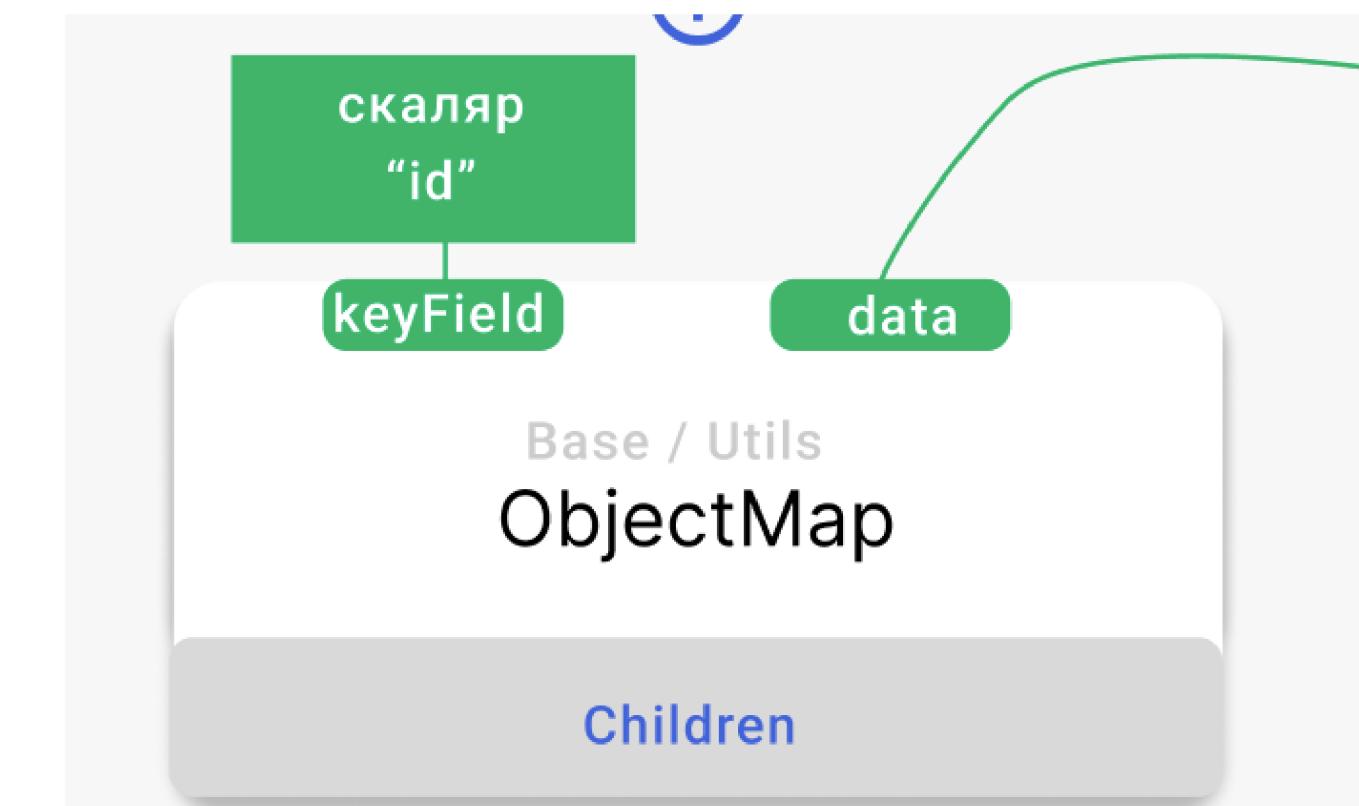


Направление данных

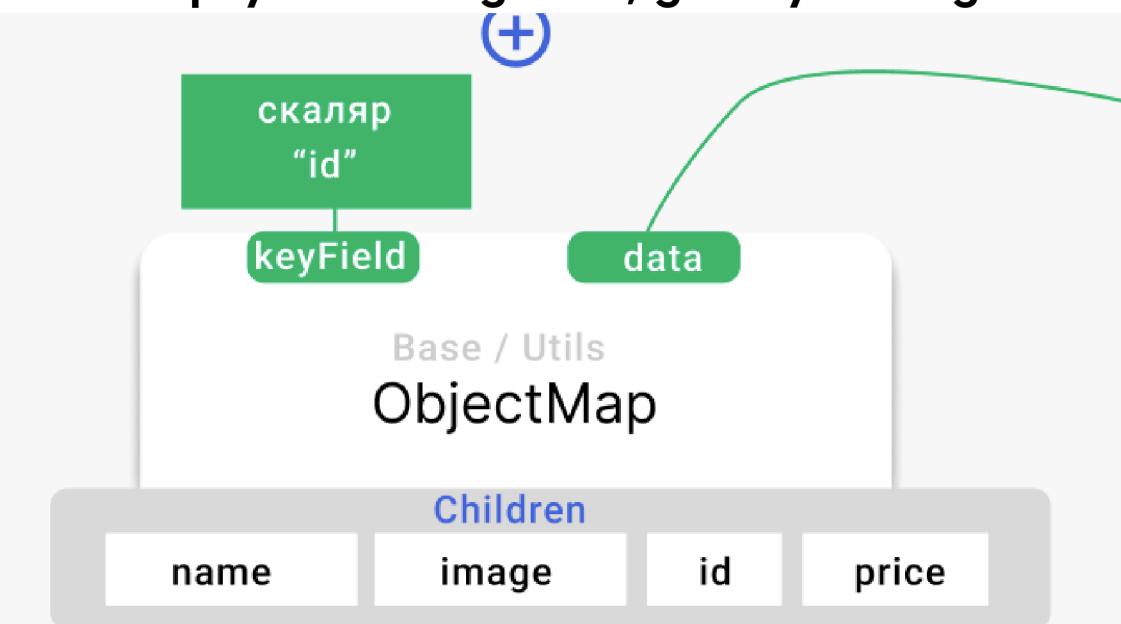


- “добавить компонент” значит добавить сверху родителя из которого можно вытащить нужные данные, но это доп. фича, не обязательная

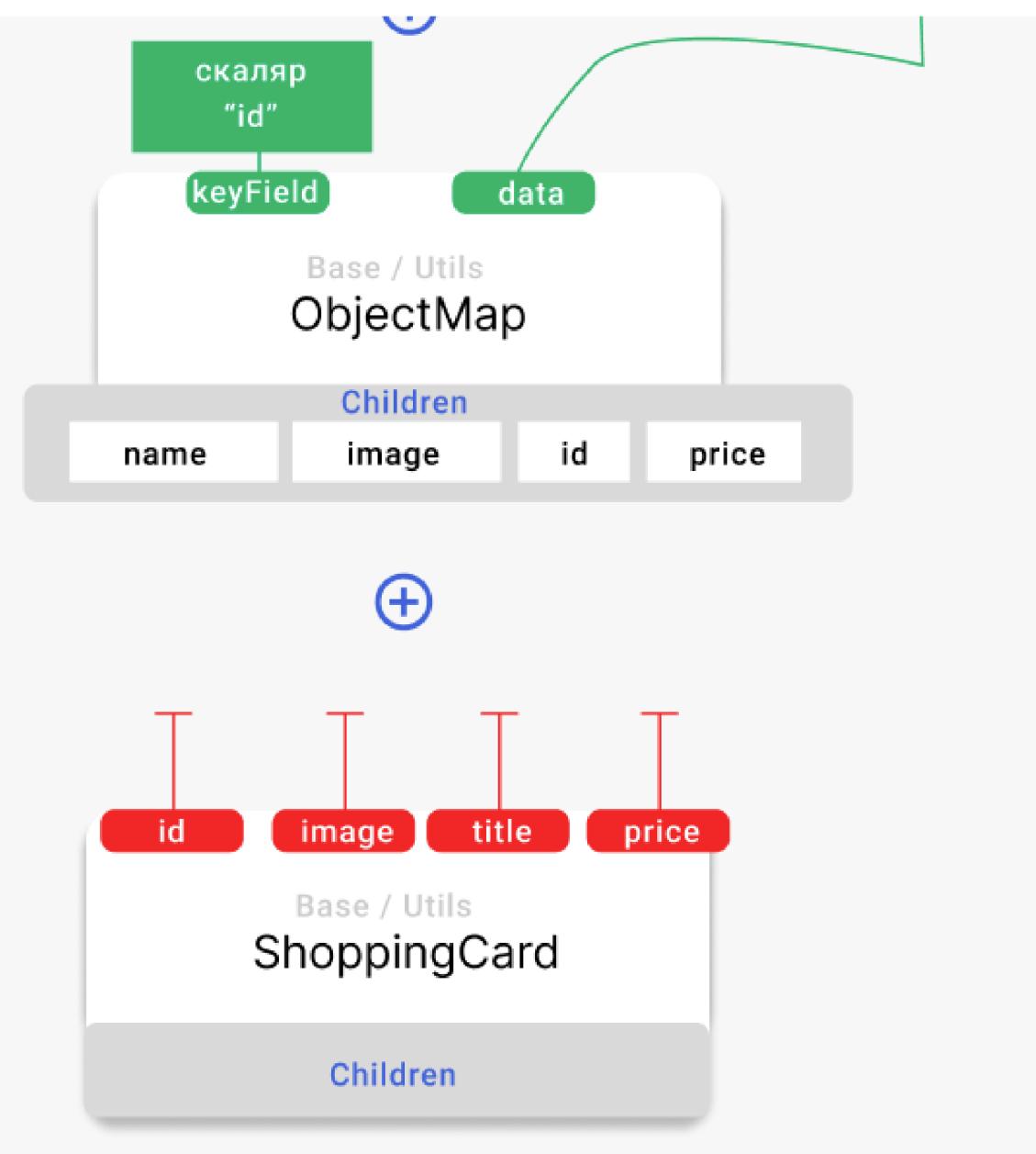
- поле “ввести реальное значение” значит ввод скаляра



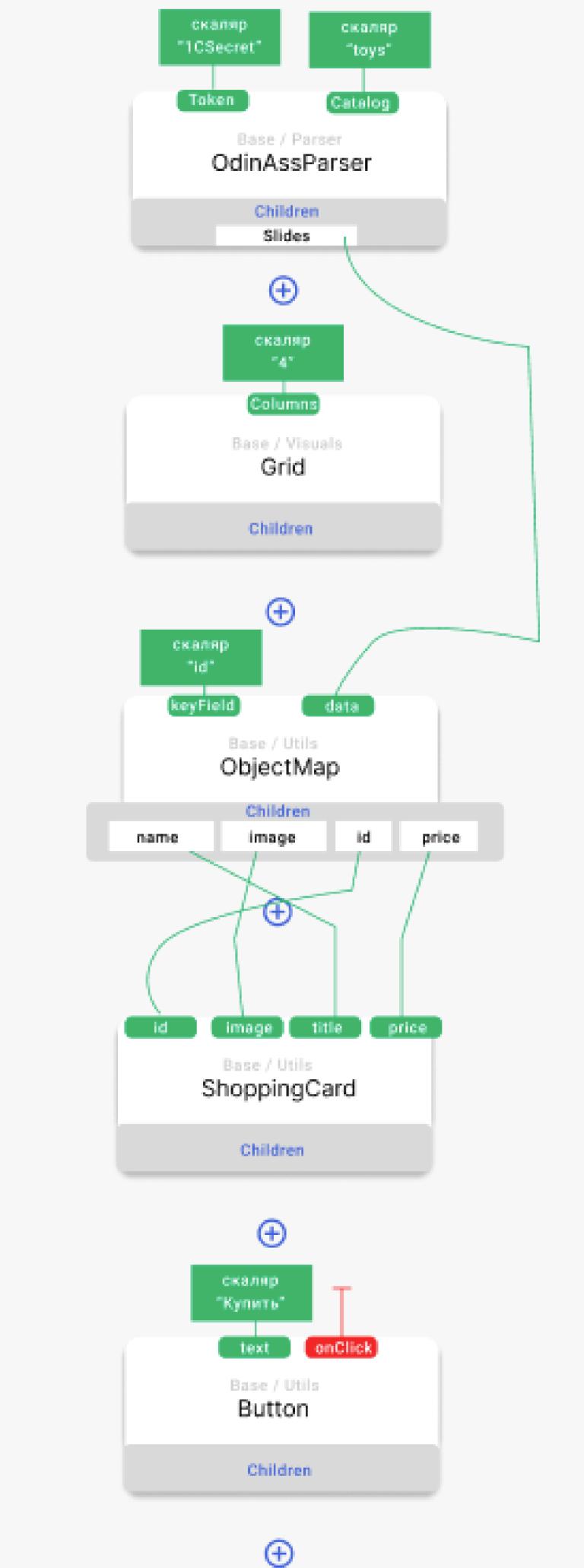
После того как генерик обновился, появляется инфа о возможных аргументахочки, доступных для использования



Направление данных



- Теперь можем соединить данные, между мапом и корзиной
- а также со всем остальным



Направление данных



- Осталась проблема с `Button` -- нужен хендлер онклика
- Допустим мы можем описать JS здесь как моковое значение
- можем добавить родительский компонент с функционалом редиректа или добавлением в корзину

Направление данных

скаляр
“Купить”

text

onClick

Base / Utils

Button

Children



Children



скаляр
“Купить”

text

onClick

Base / UI
Button

Children

- Осталась проблема с Button -- нужен хендлер онклика
- Допустим мы можем описать JS здесь как моковое значение
- можем добавить родительский компонент с функционалом редиректа или добавлением в корзину

Base / Utils

OdinAssCartAdd

Children

onClick

Children

Направление данных

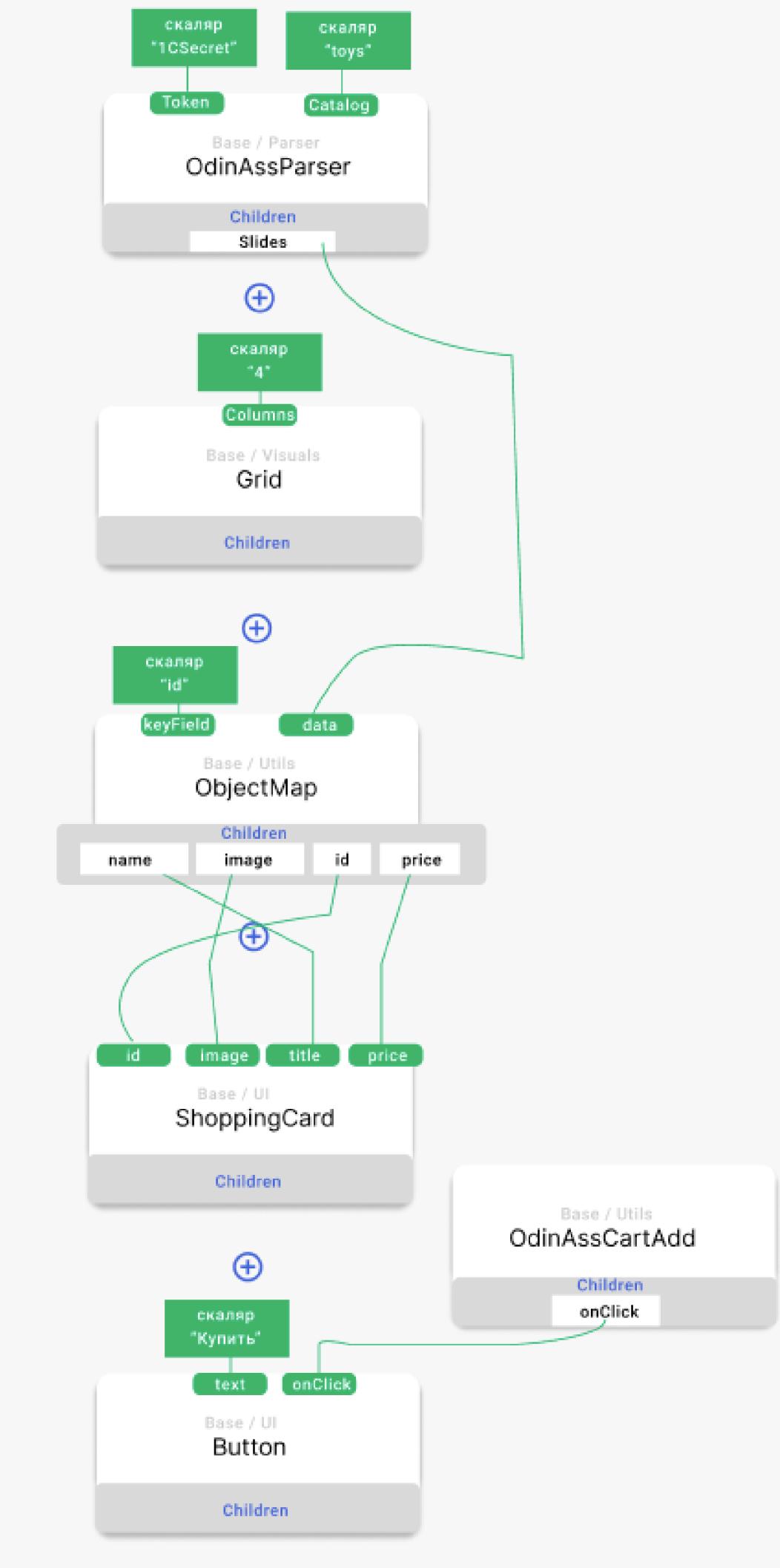
Финальная схема

Дальше пользователь сохраняет эту схему как отдельный новый компонент, доступный для переиспользования и имплементирует **уже его** на непосредственно странице проекта.

Предположим, что клиент1 запросил такой компонент в стиле “продающий красный”, мы его создали, продали, но затем клиент2 сказал, что хочет такой же, но в стиле “чарующий синий”.

Тогда зерокодер делает дубликат компонента в “красном” стиле и меняет компоненты на те что нужны, либо меняет пропы и сохраняет как новый.

Таким образом мы получили функциональность, которая выглядит по разному, работает по разному, для разных клиентов, **Вообще без привлечения кодеров** -- это очень экономно :)



Таким образом

Платформа будет обрастать новыми компонентами, как реализованными кодерами, так и зерокодерами, а мы будем их использовать для продаж, чем дольше мы будем работать, тем дешевле будут сайты, при этом

- Зеро-кодеры будут намного более “человечны” и готовы к общению с клиентом
- Мы сможем нанять их огромное количество сразу же под запросы рынка
- Полная кастомизация и гибкость при доселе невиданном количестве кодерского труда
- Based on json, поэтому огромный потенциал для использования GPT