GoCore

Фулстэк платформа для внутренней и коммерческой разработки.

Платформа GoCore (CCS.Core)

Функциональные и технические характеристики

(свидетельство о государственной регистрации программы для ЭВМ № 2019614866, от «15» апреля 2019 г).

Контакты:

Получить лицензию:
alextgco@gmail.com
t.me/alextgco

Технические вопросы:
ivantgco@gmail.com

ООО “Сложные облачные системы”
https://ccs.msk.ru
89060638866

1. Описание общей концепции

1.1. Оговорки

1.1.1. В ходе разработки ядро еще прокачалось и некоторые ранние конструкции могут отличаться от более новых, НО имеется обратная совместимость.

1.2. Основное

1.2.1. Ядро представляет собой комплекс решений необходимых для реализации различных задач для бизнеса и не только. Его задача избавить команду разработки от необходимости собирать множество различных решений и приемов для построения web приложения со всеми обязательными его частями, начиная от соединения фронта с бэком и авторизацией и продолжая продуманной ORM, умной системой кэширования, профайлом для каждой сущности, гибкой системой доступов, логированием, множеством готовых решений для различных специфических, но не редких, задач, таких как например, запросов выпадающих списков для редакторов полей или фильтров или построение данных для отрисовки дерева для иерархических данных. Со стороны фронта также имеется разработанный каркас, позволяющий с нуля подгружать меню, работать с таблицами (на основе профайлов для сущностей получаемых с бэка), формами (подвижными, масштабируемыми и сворачивающимися), встраиваемыми в формы или модальные окна (куда угодно) фреймами (не путать с iframe), также позволяющими выводить поля сущностей в соответствии с профайлами с сервера и/или реализовывать свою кастомную верстку и логику, а при желании можно встроить и отдельные например реакт приложения или компоненты и соединить их с бэком ядра.

Кроме реализации базовых потребностей web приложений, ядро содержит реализации частых бизнес процессов, таких как покупки, для которых реализованы взаимосвязанные и хорошо работающие цепочки сущностей от продуктов (товары/услуги), заказов, создаваемых для них платежей, с возможностью оплаты частями и различными методами, и отдельной сущностью транзакций платежных систем, для реализации универсального взаимодействия платежей с внешними платежными системами, до системы внутренних счетов, с пополнениями, снятиями и переводами, с архитектурой обеспечивающей целостность денежных данных. Или, например, система сбора событий (например успешных покупок) и назначения для них определенных действий, что позволяет разносить логику различных последовательных действий на независимые процессы (и при желании выносить в отдельный инстанс).

1.2.2. Ядро построено как монолит, но все общие данные, а именно кэш и блокировки, написаны так, что в любой момент могут быть вынесены во внешнее быстрое хранилище, например Redis, что позволит запускать несколько инстансов и разделять нагрузку с помощью просто увеличения количество ресурсов либо с разделением логики на различные машины.

1.3. Базовые концепции

1.3.1.1. Чтобы понимать как работать с ядром, необходимо принять несколько базовых концепций:

1.3.1.2. Понятие “Класс” в ядре это javscript класс, унаследованный от базового класса обеспечивающего загрузку профайлов соответствующей сущности в бд, связь с базой данных на основе профайлов, обеспечение прочих полезных методов. В дальнейшем может встречаться определение Сущность, которое подразумевает класс, его профайлы, методы, данные в бд, формы, таблицы на стороне фронтенда.

1.3.1.3. Профайл класса и клиентского объекта

1.3.1.3.1. Профайл содержит настройки самого класса и каждого его поля. Одной из важнейших, но далеко не единственной, задачей профайла является предоставление ядру информацию о том, как строить sql запросы к бд, включая связи между таблицами (см. пункт ниже), различные параметры по умолчанию, такие как сортировки, лимиты, наложения ограничивающих условий.

1.3.1.3.2. В отличии от классических ORM, система описания полей сущности класса предусматривает описание полей не только основной таблицы, но и полей из других сущностей, которые будут подтянуты системой через JOIN(ы). Таким образом, профайл создает что-то схожее с понятием VIEW в классических реляционных СУБД, но конечно, описание полей в профайле не ограничивается только этим, а служит еще и для многих других целей.

1.3.1.3.2.1. Это базовая концепция. Когда разработчик проектирует сущности для какого-то решения, поля из связанных таблиц. закладываются сразу в класс. Например в классе user есть не только поле country_id, но и country_name, и country_code, которые являются

виртуальными, то есть подтягиваются из другой таблицы (ds_user_country), по ключу country_id. То есть эти связи заложены в профайле, а не строятся разработчиком во время запроса. Разумеется при запросе, пользователь может ограничить набор полей, который ему необходим и лишние JOIN(ы) собираться не будут.

1.3.1.3.2.2. Первичное создание профайла и полей осуществляется в файле tables.json, после чего может модифицироваться через интерфейс системы. В tables.json описывается только самое необходимое, а большинство настроек имеют оптимальные значения по умолчанию.

1.3.1.3.3. Автоматически создаваемые поля. Все таблицы автоматически имеют системные поля, такие как, created(datetime), updated(datetime), deleted(datetime) (ядро работает по принципу soft deletion), created_by_user_id, last_edited_by_user_id, deleted_by_user_id и их виртуальные поля для подтягивания фио, также еще некоторые системные, для различных механизмов. Кроме этого для таблиц с иерархической связью, автоматически создаются поля node_deep, parents_count, children_count, корректность содержимого которого поддерживается ядром автоматически. Кол-во родителей и детей в данных полях указано именно на всю глубину, а не только на один уровень (прямой потомок/родитель).

1.3.1.3.4. Кроме функций построения запросов, и других взаимодействий с бд, таких как например, можно ли создавать записи в этой таблице, или при создании (или при редактировании) указывать значение определенного поля и прочих, профайл содержит различные настройки, большая часть из который используется на стороне фронтенда для определения внешнего вида и функциональности компонентов. Тут также есть настройки всего класса и настройки каждого поля. Пример настроек класса - количество записей на страницу (для построения пагинации в таблицах), имя формы, которую надо открыть из контекстного меню таблицы, заголовок. Примеры настроек для полей: Наименование, текст

подсказки, тип редактора, выводить ли фильтр по нему и какой тип этого фильтра, минимальное и максимальное значение для полей числового типа и минимальный шаг для изменения стрелочками, и прочее. См. Профайл класса/клиентского объекта и их полей.

1.3.1.3.5. Клиентские объекты.

1.3.1.3.5.1. Это копии класса, но со своим профайлом его самого и его полей. При этом набор полей тот же. Это необходимо в основном для фронтенда, чтобы в таблице например запретить создание, а в форме разрешить, или в разных таблицах выводить разный набор полей и, например, одну таблицу сделать для одной роли, а другую для другой.

1.3.1.3.5.2. На стороне фронтеда клиентские объекты обычно связаны с таблицей, формой или фреймом, которые опираясь на профайл и верстку (актуально для форм и фреймов), создают компоненты на странице. При этом у них может быть свой кастомный js код, позволяющий по разному взаимодействовать с данными и вызывать различные методы.

1.3.1.4. Большая часть логики разрабатываемого приложения, должна располагаться в методах вышеупомянутых классов. Точкой входа в любой процесс является какой либо метод определенного класса (через внутреннее api см. ниже).

1.3.1.4.1. Могут быть какие-то внешние механизмы, например интеграции с платежными системами, где реализация интеграции с конкретной системой вынесена в отдельное место: /modules/payment_system/instances/<payment_system_name>, но при этом конкретный инстанс переопределяет или дополняет метод универсального класса платежной системы в /modules/payment_system/instances/index.ts методы которого вызываются из методов сущности Ps_transaction (класс транзакций с внешними платежными системами). На примере интеграций с платежной системой, есть еще вторая точка входа, когда платежная система обращается на url системы для оповещения об успехе или неудаче проведения

оплаты, и тут у нас имеется /routes/index.ts где обрабатывается ответ, перенаправляется также в Ps_transaction.result, который вызывает завершение в нужном инстансе платежной системы, в ее метод result, который в свою очередь (после парсинга, проверок подписи) вызывает завершающий метод super.result родительского класса интеграций платежных систем (/modules/payment_system/instances/index.ts), а он сохраняет результат средствами того же внутреннего api.

1.3.1.4.1.1. Этот пример показывает, что различные внешние модули, занимаются лишь специфичной для них работой, а далее управление снова передается в основные методы сущностей.

1.3.1.5. Все методы, как других, так и того же классов должны вызываться через внутреннее API.

1.3.1.5.1. Иногда, в рамках метода, если нужно вызвать метод того же класса можно вызвать его через this, но надо отдавать себе отчет в том, что в таком случае, вызываемый метод не будет помещен в цепочку вызова и не будет отдельно залогирован, блокировки и снятия блокировк будет работать, но снятие произойдет только при завершении именно исходного метода, который прошел через api (если блокировки имеются). Лучше избегать таких вызовов, если только вы не хотите избежать излишнего логирования и точно понимаете что делаете.

1.3.1.5.2. Для обращения же к методам других классов всегда следует использовать внутреннее api, так как для работоспособности недостаточно просто создать новый инстанс и вызвать его метод.

1.3.1.5.3. Подробнее, что происходит во внутреннем api смотри в ”Внутреннее API. Подробно”.

1.3.1.6. Не используем enum, используем справочники вида ds_ (отдельно смотри про договоренности наименований).

1.3.1.6.1. Запросы можно строить по sysname, ядро самостоятельно возьмет id из кэша или сходит в базу и подставит его в оригинальный запрос.

1.3.1.6.2. Создание/изменение записи тоже может быть с sysname, а не по id, ядро также сделает всю работу за вас.

1.3.1.6.3. Подытоживая, вы свободно можете работать по sysname не думая о том, что в базу надо подставить сопоставленный номер.

1.3.1.6.4. Все sysname в справочниках пишем UPPERCASE. Так как они сразу выделяются в коде. В дальнейшем, возможно стоит доработать ядро так, чтобы содержимое справочника автоматически синхронизировалось в ts типы в соответствующем файле структур класса, тогда можно будет использовать не строку, а что-то вроде "ds_order_status.CREATED". Это повысит контроль за кодом.

1.3.1.7. Входные и выходные параметры каждого метода стандартизированы.

1.3.1.7.1. Входным параметром любого метода любой сущности является объект запроса "r" соответствующий интерфейсу IAPIRequest, который содержит информацию о цепочке, клиенте, конкретном подключении, системных параметрах запроса, классе и методе запроса и параметрах метода.

1.3.1.7.2. Любой метод должен возвращать объект соответствующей интерфейсу IAPIResponse (иногда вы можете увидеть что выходной параметр должен соответствовать IError, это устаревший вид, но IAPIResponse расширяет IError и по факту, все ответы соответствуют что первому, что второму).

1.3.1.7.2.1. Для соответствия этому интерфейсу используется один из трех классов ошибок/успеха: UserOk (используется для успешных ответов), UserError (для пользовательских ошибок, таких как, например, "Товар закончился" или "Пароль неверный"), MyError (используются для внутренних ошибок, типа неверно переданные параметры (не пользовательские, а те, что передаются кодом, например, "Не передан id" (пользователь сам никогда не вписывает id, поэтому это ошибка относится к неправильному использованию метода)).

1.3.1.7.2.2. Все эти классы имеют поле code, которое только для UserOk будет равно 0, а для остальных отличное от 0 значение. Для пользовательских методов, по умолчанию,

для ошибок вида MyError и UserError код будет одинаковый (-1000), а особые коды используются лишь для некоторых особых условий, которые преимущественно используются для механизмов ядра. Например, если запрос неавторизован или сессия просрочена код будет -4, а для недостатка доступа используется код 11. Но повторюсь, кодами пользуются лишь некоторые механизмы ядра, и при разработке логики они не используются (кроме проверки на 0). Вы можете посмотреть существующие коды в /error/error.js (многие могут быть не актуальны).

1.3.1.7.2.3. Ответ может выглядеть так:

```
return new UserOk('Все хорошо, пейлоад лежит в data',
{calculatedAmount: 10, hint:'Посчитано для 5 строк'})
```

1.3.1.7.2.3.1. Тогда эти данные можно будет получить в res.data. Например console.log(res.data.calculatedAmount) // выведет 10.

1.3.1.7.2.4. Ответ с ошибкой также может содержать payload (поле data), в который имеет смысл добавить и выше возникшую ошибку, и параметры, с которыми был вызван вышестоящий метод, вызвавший ее.

1.3.1.7.2.4.1. Имеется два пути обработки такой ошибки.

1.3.1.7.2.4.1.1. Или передать ее как есть, возможно дополнив и/или изменив текст, но сохранив инстанс и соответственно тип ошибки:

```
if (res.code) return res
```

1.3.1.7.2.4.1.1.1. Есть методы для дополнения объекта ошибки. setMsg и setData. setMsg принимает новый текст и опционально, вторым параметром объект, для дополнение data. Второй метод

setData принимает только объект для дополнений data.

```
// get row
p = {id: params.id}
res = await r.api(Deal, 'getById', prepareP(p))
if (res.code) {
   // return res
   // return res.setMsg('Новый текст ошибки', {p})
   return res.setData({p})
}
```

1.3.1.7.2.4.1.2. Или создать новую, с новым текстом и, возможно, новым типом, и в data указать ошибку и параметры.

```
p = {id: params.id}
res = await r.api(Deal, 'getById', prepareP(p))
if (res.code) {
   return new UserError('Возможно кто-то удалил документ, обновите
таблицу и попробуйте еще раз.',
       {res, p})
}
```

1.3.1.8. Никаких throw.

1.3.1.8.1. Концепция ядра такова, что методы никогда не кидают ошибку, а вместо этого возвращают объект UserError или MyError. Этого принципа и следует придерживаться при написании собственных методов.

1.3.1.8.1.1. Если метод использует какую-либо встроенную или стороннюю функцию, которая может выбросить ошибку, то ее следует обернуть в try/catch и в catch вернуть инстанс IAPIResponse.

1.3.1.8.1.2. Естественно, на верхнем уровне внутреннего api, а также еще выше, на уровне express имеется ловушка для непредвиденных ошибок, но это аварийный механизм.

1.3.1.8.1.3.

1.3.1.8.2. Отсюда вывод, всегда проверять res.code.

1.3.1.8.2.1. Может показаться, что данный подход усложняет написание кода, за счет необходимости делать эту обработку после вызова каждого метода через внутреннее api,

однако взамен мы приобретаем следующие преимущества:

1.3.1.8.2.1.1. Классический Exception предоставляет стек, но с учетом всех оптимизаций/минификаций, вызовов через анонимные функции и в целом очень большого количества промежуточных функций, читать этот стек практически невозможно, а его логирование обычно срезает большую часть данных. Механизм используемый в ядре предоставляет четкую цепочку вызова методов и ответов на каждом из уровней если они были уточнены, то с доп данными. То есть мы знаем, что метод один был вызван с такими то параметрами и вернул такой то ответ, внутри он вызвал следующий метод уже с другими параметрами и его ответ также имеется, и так далее.

1.3.1.8.2.1.2. Второе, преимущество (по мнению создателей ядра), состоит в том, что итак обрабатывать ошибки нужно на каждом уровне вызова (то о чем говорится, в предыдущем пункте), но с выбросом Exception это превращается в громоздкие try/catch вместо короткого

```
if (res.code) return res.setData({p})
```

или более полного:

```
if (res.code) return res.setMsg('Новый текст', {p,
anotherAdditionalInfo:{abc:123}})
```

1.3.1.9. Принцип написания любого метода.

1.3.1.9.1. Перед методом необходимо оставлять документирующий комментарий, достаточный для понимания, что именно делает метод и какие имеет особенности. Параметры при этом в нем описывать не требуется, так как стиль JSDoc создаваемый автоматически укажет только параметр r, а разработчика интересует только содержимое r.data.params. Они будут описаны с помощью generic, см. пункт ниже.

1.3.1.9.2. Входные и выходные параметры, а именно их определяющая для конкретного метода часть документируется в generics для IAPIRequest и IAPIResponse соответственно. Также там можно указать комментарий для параметров, в однострочном или многострочном стиле. В коде ниже можно будет увидеть пример.

1.3.1.9.3. Код метода, представляет собой условно 3 блока.

1.3.1.9.3.1. Проверка входных параметров, и определение переменных уровня метода, которые пригодятся далее.

1.3.1.9.3.1.1. Параметры метода расположены в r.data.params. Параметры запроса (нужны реже, так как это скорее системные параметры и используются ядром) расположены в r.params. Подробнее см. описание IAPIRequest.

1.3.1.9.3.1.2. Как правило в начале определяются 2 переменные p и res, которые будут многократно переиспользованы для вызовов методов через внутреннее api.

1.3.1.9.3.2. Основная логика состоящая из последовательного вызова необходимых методов, проверок и вычислений.

1.3.1.9.3.2.1. Как правило это получение данных, какие нибудь проверки, вызов других методов, сравнения, синхронизации данных, генерация чего-либо, сохранение изменений в базу.

1.3.1.9.3.2.2. Многие разработчики любят создавать отдельный метод под каждую микрозадачу, однако мы придерживаемся концепции, что выносить логику следует в первую очередь, в том случае, если она используется несколькими методами или если метод или файл сильно разрастаются.

1.3.1.9.3.3. Возврат успеха.

1.3.1.9.3.3.1. Этот блок весьма условный, так как метод может завершиться и раньше при определенных условиях, как успехом, так и не удачей. Но в конце в любом случае должен быть конечный return.

1.3.1.9.4. Пример метода:

```
/**
* Method will update the description of the deal by setting the current time,
* after which it will prepare the pdf
*/
async doDoc(r: IAPIRequest<{
   id: number
}>): Promise<IAPIResponse<{
   timeOfOperation?:string // Time that will be added to the description
   /*
   * Url to the generated pdf
    */
   pdfLink:string
}>> {
   const rParams: IAPIParams = r.params
   const params: IAPIQueryParams = r.data.params

   const {id} = params
   if (isNaN(+id)) return new MyError('id not passed')

   let p
   let res: IAPIResponse

   let pdfLink = ''
   const timeOfOperation = toUserFriendlyDateTime()

   // get row
   p = {
       id,
       columns:['id', 'status_sysname'],
   }
   res = await r.api(Deal, 'getById', prepareP(p))
   if (res.code) {
       return new UserError('Maybe someone deleted the deal, update the table and
try again.', {res, p})
   }
   const row = res.data.rows[0]

   if (row.status_sysname !== 'READY') {
       return new UserError('The deal is not ready yet', {row})
   }

   // Update the deal description
   p = {
       id:row.id,
       description: `Test_description_${timeOfOperation}`,
   }
   res = await r.api(Deal, 'modify', prepareP(p))
   if (res.code) res.setMsg('Failed to update the deal description', {p})

   // Creating pdf...

   return new UserOk('Pdf created', {timeOfOperation, pdfLink})
}
```

1.3.1.9.5. Выделение методов в отдельный файл. Ключевые сущности проекта, как правило имеют большое количество методов и их стоит выносить в отдельные файлы, разбивая по логике. Файлы сущностей располагаются в /classes/. Например /classes/Deal.ts. Часть методов можно вынести в отдельные файлы и расположить их следует в

одноименной директории (ее нужно создать) /classes/Deal/someMethods.ts

1.3.1.9.5.1. Так как выделяемые в отдельный файл методы не должны терять доступа к контексту, то его следует передавать. Один из способов:

1.3.1.9.5.1.1. Метод в отдельном файле может выглядеть так:

```
export async function prepareReceipt(this: Payment, r: IAPIRequest<{
   ids: number[] // ID платежей. Можно указать несколько - объединит в один чек
}>): Promise<IAPIResponse<{
   receiptItems:{
       name:string
       price: number
       quantity: number
       amount: number
   }[]
}>> {
…
```

1.3.1.9.5.1.2. А в самом классе делаем только определение метода и вызов вынесенного через call

```
/**
* Подготовит элементы платежа для чека. Метод учитывает, что для позиции заказа, могут быть указаны
* специальные подэлементы для чека. Метод приведет стоимость таких подэлементов к цене в платеже.
*/
async prepareReceipt(r: IAPIRequest<{
   ids: number[] // ID платежей. Можно указать несколько - объединит в один чек
}>): Promise<IAPIResponse<{
   receiptItems:{
       name:string
       price: number
       quantity: number
       amount: number
   }[]
}>> {
   return await prepareReceipt.call(this, r)
}
```

1.3.1.10. CRUD. В ядре это AddGetModifyRemove.

1.3.1.10.1. Любой класс имеет эти четыре метода (и не только их) для работы с базой данных.

1.3.1.10.2. Подробнее описано в “Методы базового класса”.

1.3.1.11. Не залезаем в БД. Не пишем запросы руками.

1.3.1.11.1. В БД через сторонние программы разработчику может потребоваться, в большинстве случаев, залезть только для того чтобы сделать или загрузить дамп. Изредка может потребоваться залезть, чтобы изучить данные при какой-то отладки. И, совсем редко, может потребоваться манипуляции со

структурой или данными, если в процессе разработки были допущены крупные ошибки, и хочется переделать некоторую часть, а механизмы ядра не справляются в связи с каким-нибудь подвисшими данными или чем-то еще.

1.3.1.12. Договоренности по наименованиям, синтаксису, прочему

1.3.1.12.1. Наименование справочников d_ и ds_

1.3.1.12.1.1. Ядро предусматривает справочники двух типов.

1.3.1.12.1.1.1. Это системные справочники “ds_”, от слов dictionary system. Используются для таких справочников, которые могут меняться только в процессе разработки, но не в процессе эксплуатации. Например ds_order_status или ds_payment_type

1.3.1.12.1.1.2. Справочники вида “d_”, от слова dictionary. Могут меняться в процессе эксплуатации системы. Например d_product_category.

1.3.1.12.1.1.3. Иногда граница может быть размыта, например language, может быть как d_ так и ds_, так как справочник может быть сразу полностью (достаточно) залит или пополняться по мере надобности.

1.3.1.12.1.1.4. Такое наименование помогает при миграции данных, где справочники ds_ всегда переносятся, а d_ могут быть перенесены разово или вообще не переносится. Кроме этого это позволяет визуально лучше ориентироваться в таблицах, если требуется глубокая отладка или опять таки мердж данных. Еще такие сущности, автоматически попадают в меню Справочники или Системные справочники.

1.3.1.12.2. Типы полей. Используется в tables.json при описании.

1.3.1.12.2.1. id - bigint(20)

1.3.1.12.2.1.1. Часто можно увидеть что bigint используется для id даже маленьких справочников, например ds_gender. Мы

ранее не уделяли этому внимания, однако лучше использовать наиболее подходящий тип. См.в разделе, как заполнять tables.json

1.3.1.12.2.1.2. Вы можете указывать и другие размерности, но в этом случае не забывайте, когда вы в другой сущности используете поле как вторичный ключ к первой сущности, оно должно быть того же типа.

1.3.1.12.2.2. Чекбокс - tinyint(1)

1.3.1.12.2.2.1. Для булевых полей всегда использовать этот тип и длину. Именно так предусмотрено в MariaDB и именно на этот показатель ориентируется ядро для различных действий.

1.3.1.12.2.3. Остальные типы соответствуют типам MariaDB.

1.3.1.13. Бэкофис и фронтенд. Как организовать

1.3.1.13.1. Варианты реализации

1.3.1.13.1.1. Во многих проектах основной фронтенд ядра является как бэкофисом, административной панелью и интерфейсом для конечного пользователя. Во фронтенде ядра сразу имеются все интерфейсы, как для разработки (управление профайлами, меню, доступами, системными справочниками…), так и для администратора системы (опять таки

управление пользователями и их доступом, настройками относящимися к проекту), а также для конечных пользователей (в рамках конкретного проекта строятся интерфейсы для конечного пользователя).

1.3.1.13.1.1.1. В этом случае, это, в большей степени, одностраничное приложение.

1.3.1.13.1.1.2. Именно такая реализация имеется из коробки.

1.3.1.13.1.2. Второй вариант, писать фронтенд проекта отдельно, будь то сайт, ERP-система или что-либо еще. Тогда бэкофис остается для разработчика и администраторов системы(опционально, так как и для них можно написать отдельный фронтенд), а фронтенд конечного пользователя пишется отдельно (на любом фреймвроке или нативно) и общается с бэкэндом по API (про типы подключения читай ниже).

1.3.1.13.1.3. Еще вариант писать мультистраничное приложение (сайт) с рендерингом на стороне сервера (SSR), используя роутинг предоставляемый фреймворком express, который входит в часть ядра, и использовать движок шаблонизации, например jade (pug).

1.3.1.13.1.4. Также можно скомбинировать первый и третий подходы и написать фронтенд на отдельном инстансе ядра используя SSR, это позволит инстансу “сайта” заниматься только сайтом, обеспечивая соединение к бэкэнду (отдельный инстанс, где реализована основная логика проекта).

1.3.1.13.1.5. Для мультистраничных решений, а также других запросов извне используется стандартный роутинг фреймворка express, откуда можно использовать внутренний API. Подробнее см. роутинг.

1.3.1.13.2. Подключение к бэкэнду

1.3.1.13.2.1. Встроенный фронтенд использует нашу библиотеку для подключения доступную в npm: go_core_query (https://www.npmjs.com/package/go_core_query), которая обеспечивает непрерывное соединение с сервером по сокету, умеет

обрабатывать ответ об не авторизированном запросе (действие сессии истекло) и перенаправлять на страницу авторизации (можно переопределить это действие). Предоставляет два метода общения с сервером: через асинхронную функцию api (максимально схожую с серверным внутренним api); и конечно общение через emit/on - то есть все то, что предоставляет socket.io. Большая часть логики обычно реализуется через api, так как архитектура приложения значительно прозрачнее, когда есть запрос и есть ответ, но для некоторых задач можно использовать сокет как есть (для обеспечения двусторонней связи).

1.3.1.13.2.1.1. Функция api, кстати, также работает через socket, а не через fetch, но разработчику об этом думать не надо.

1.3.1.13.2.1.2. Для того, чтобы в консоле видеть запросы и ответы, необходимо включить режим отладки: вызовите в консоле (браузера) debug(); после чего вы должны увидеть информацию об изменении режима отладки; обновите страницу.

1.3.1.13.2.1.3. В своих приложениях вы также можете использовать эту библиотеку, как на фронте так и на бэкэнде, настроив ее под свои нужды. Подробнее см. документацию на библиотеку.

1.3.1.13.2.2. Кроме библиотеки вы можете использовать простой http(s) API, через fetch или другие механизмы. Вам потребуется выполнить запрос авторизации, в ответ вы получите JWT, который требуется добавлять в Authorization header: Authorization: Bearer <Token>, к остальным запросам. Если сессия окажется просроченной или принудительно обнулена, то в ответ на любой запрос вы получите стандартный объект ответа (IAPIResponse) с кодом “-4”, который говорит вам о том, что следует авторизоваться заново.

1.3.1.13.3. Мобильное приложение

1.3.1.13.3.1. Мобильное приложение может быть написано на чем угодно, в частности на ReactNative. Для RN приложений также как и для обычного фронтенда доступна библиотека для подключения. В остальных случаях вам подойдет http(s) API. Подробнее

2. Внутреннее API. Подробно.

2.1. Что происходит внутри

- 2.1.1. Проверка параметров
  - 2.1.1.1. Приведение к стандарту (класс с Большой буквы)
  - 2.1.1.2. размер, корректность
- 2.1.2. Первичная Загрузка параметров из DB (выполняется один раз)
- 2.1.3. Проверки доступов
- 2.1.4. Логирование
- 2.1.5. Создание цепочки запроса
- 2.1.6. Создание, пролонгирование роллбэк ключа
  - 2.1.6.1. Проведение роллбэка
- 2.1.7. Defer функции
- 2.1.8. Стекование одинаковых запросов (постановка в очередь, пока не выполнится первый, а потом остальные вызываются (из кэша работают быстро)).
  - 2.1.8.1. Можно доработать и возвращать один всем тот же ответ, но тогда стекать нужно с учетом пользователя.
- 2.1.9. Отлов
  - 2.1.9.1. Цикличных запросов
  - 2.1.9.2. Вывод в консоль секретных параметров, таких как пароли - скрыт
  - 2.1.9.3.
- 2.1.10. Получение инстанса (Синглтон (для каждого клиентского объекта свой)) класса
  - 2.1.10.1. С ожиданием инициализации при одновременном запросе
- 2.1.11. Выполнение действия
- 2.1.12. Обработка результата
  - 2.1.12.1. Проверка размера
  - 2.1.12.2. Роллбэк
    - 2.1.12.2.1. Откат
    - 2.1.12.2.2. Сохранение отката (можно откатить позже)
  - 2.1.12.3. Снятие блокировок установленных внутри методов
  - 2.1.12.4. getUserErrFromErrRes
    - 2.1.12.4.1. Всплытие наверх UserError, вместо MyError
  - 2.1.12.5. noAuth
  - 2.1.12.6. Логирование ошибок в базу
  - 2.1.12.7. Фиксация времени

2.2. Как методы вызываются внутри

2.2.1. Почему через API

2.2.1.1. Создание инстанса. Синглтон

2.2.1.1.1. При создании инстанса происходит загрузка профайла класса и его полей, которые влияет на построение запросов, при взаимодействии с базой данных, формировании запрашиваемых данных, а также могут иметь другие специфические настройки для конкретного класса.

2.2.1.1.2. На самом деле это не совсем синглтон, так как при обращении к отдельному клиентскому объекту (отдельной копии профайла и полей, которая может иметь отличные от исходного профайла настройки), создается отдельный инстанс. То есть, один <класс>.<клиентский объект|null> = один инстанс.

2.2.1.2. Цепочка запросов, откаты, разблокировки, выполнение defer функций, логирование, обработка ошибок.

2.3. Rollback

2.3.1. Как работает, почему не на уровне БД

2.3.1.1. Необходимо доработать и перенести на уровень бд. То есть использовать механизм транзакций.

2.3.1.1.1. Сделать несложно, надо в getConnP() прокинуть параметр rollback_key, создать стек пулов по ключу, создавать или возвращать уже созданный с таким ключем. Сделать два метода rollbackTransaction и commitTransaction с ключем. Вызывать их в api по завершении цепочки. Заблокировать запросы для старого механизма роллбэка.

2.3.1.1.2. Отличие существующего механизма в том, что можно сохранить результат успешного выполнения и откатить его спустя время.

2.3.1.1.2.1. Можно сохранить этот функционал для некоторых запросов, для которых будет параметр save.

2.3.1.2. В текущей версии не требуется никаких дополнительных действий, чтобы в случае ошибки в одной из функций произошел откат всей цепочки изменений до этого (до точки входа в цепочку, например, запрос с клиента). Однако в предыдущих версиях требовалось явно указать rollback_key в параметрах запроса на изменение/добавление/удаление или другой кастомный метод, а также в начале функции определить этот rollback_key, получив из запроса выше или создав. Кроме этого в конце функции необходимо было в случае ошибки вызвать rollback, а в случае успеха, при желании, сохранить цепочку отката. Этот код продолжает

работать, однако он более не нужен, так как это все выполняется автоматически в api.

2.3.2. Как управлять цепочками роллбэков.

2.3.2.1. Как было указано выше, по умолчанию, все будет работать самостоятельно, однако поведением можно управлять. Вы можете внутри какого-нибудь метода, вместо продолжения текущей цепочки, начать новую цепочку. Это может пригодиться, когда вы итерируете набор, и для каждого элемента хотите провести независимый набор действий. То есть для каждого своя транзакция.

2.3.2.1.1. Чтобы начать новую цепочку нужно в параметрах передать **newRollbackBranch**:true. Это параметр самого запроса (r.params), а не параметры метода (r.data.params), но вы можете указывать его в параметрах метода, так как ядро само перенесет этот параметр в r.params. Пример:

```
const userIds = [1, 2, 4]
for (const userId of userIds) {
   p = {
       id: userId,
       newRollbackBranch: true
   }
   res = await r.api(User, 'confirmUser', prepareP(p))
   if (res.code) return res
}
...
```

2.3.2.2. Кроме начала новой цепочки, вы можете указать, что хотите сохранить цепочку действий, в случае успеха, для возможности ручного отката этих изменений.

2.3.2.2.1. Для этого необходимо установить параметр запрос saveRollback в true, в коде функции. Пример: в самом начале функции напишите r.params.**saveRollback** = true:

```
async exampleMethod(r: IAPIRequest<{
   id: number,
   toDateTime: string // 'DD.MM.YYYY HH:mm:ss'
}>): Promise<IAPIResponse> {

   r.params.saveRollback = true
   ...
```

3. Базовые структуры (типы и интерфейсы)

3.1. В системе имеется определенное количество базовых структур (интерфейсов/типов), которые определяют различные форматы данных,

параметры вызова функций и их выходные параметры. Они преимущественно хранятся в двух файлах: libs/api/APIStructures.ts, структуры, преимущественно относящиеся к внутреннему API, и models/system/CoreClass/structures.ts, относящиеся к базовому классу (CoreClass), его методам и базовым механизмам.

3.2. Структуры имеют комментарии там где это необходимо.

4. Базовый класс

4.1. Инициализация класса

4.1.1. Загрузка профайла

4.2. Методы базового класса

4.2.1. CRUD

4.2.1.1. Описание

4.2.1.1.1. Данная аббревиатура не подходит к наименованию методов в ядре, но здесь будут описаны методы аналогичные, но с другими названиями. Add/Get/Modify/Remove.

4.2.1.1.2. На уровне CoreClass методы обращаются к своим прототипным методам <method_name>Prototype (например get и getPrototype). Это позволяет в отдельных классах при переопределении метода, например, get иметь возможность вызвать исходный метод без использования super или .prototype. Это нужно, чтобы родительский метод можно было вызвать через внутреннее API, так как это основная концепция ядра.

4.2.1.1.2.1. Например, после переопределения modify у Product, вы можете вызвать неизмененный? родительский modify через внутреннее API следующим образом:

```
p = {
   id: 100,
   name: 'new_product_name',
}
res = await r.api(Product, 'modifyPrototype', prepareP(p))
if (res.code) return res
```

4.2.1.1.2.2. .

4.2.1.2. get

4.2.1.2.1. Получает на вход условия выборки. Возвращает набор записей в массиве rows (res.data.rows). Работает через кэш.

4.2.1.2.2. Входные параметры. Интерфейс IAPIQueryParams

4.2.1.2.2.1. columns

```
columns?: string[]
```

4.2.1.2.2.1.1. Массив строк, с перечислением колонок, которые должны быть в результирующем ответе. Система проверит наличия такого поля в профайле и не позволит запросить несуществующее поле или подставить туда SQL выражение.

4.2.1.2.2.1.2. Если не передавать, то будут возвращены все поля, для которых установлена галочка visible в профайле Класса или Клиентского Объекта.

4.2.1.2.2.1.3. as

4.2.1.2.2.1.3.1. Имя поле может быть указано с использованием “as”, тогда указанное поле будет возвращено в колонку с указанным именем. Пример: columns: [‘user_id as id’]. Тогда будет возвращено поле user_id в колонку id. Это может понадобится, когда данный запрос формируется для использования в другом запросе, см. whereInSelect (он опирается именно на id и формирует запрос вида where <field_id> in (select **id** from …)).

4.2.1.2.2.2. where/param_where. Условия выборки.

4.2.1.2.2.2.1. По умолчанию нет условий (только применяется limit по умолчанию см. limit).

4.2.1.2.2.2.1.1. Для класса или клиентского объекта может быть определен default_where, который будет наложен на все запросы к этому классу/КО.

4.2.1.2.2.2.2. where это массив объектов условий (IOneWhere), где одно условие может

быть, либо непосредственно условием, с указанием поля, типа сравнения и значение(я), либо группой условия, с типом сравнение между элементами и items - таким же массивом условий. Нет необходимости запоминать все параметры приведенного ниже интерфейса, так как для формирования объекта имеются удобные функции обертки (о них ниже).

```
export interface IOneWhere {
   key?: string,
   type?: string
   val1?: any
   val2?: any
   group?: string
   comparisonType?: string
   is_role?: boolean,
   role_group?: string
   name?: string // info
   isGroup?: boolean // if set, items must be an array
   items?: IOneWhere[]
   // itemsSQL?:string
   isOr?: boolean // Определяет какой тип сравнения будет с предыдущем элементом.
   // Если эта группа, то через OR будет соединена вся группа
   binary?: boolean
   s?: string // sql string for one where or for all items (if isGroup)
}
```

4.2.1.2.2.2.1. .

4.2.1.2.2.3. Функции обертки.

4.2.1.2.2.3.1. Все функции реализованы в\models\system\CoreClass\prepareWhere.ts и имеют документирующий комментарий. Однако опишем их и здесь. Задача таких функций, получить на вход необходимые для составления условия параметры, а на выход вернуть объект соответствующий IOneWhere. Все функции помимо обязательных могут принимать последним необязательным параметром объект params, и тогда он будет развернут в результирующий объект (в конце).

4.2.1.2.2.3.2. **whereEq, whereNotEq, whereMore, whereMoreEq, whereLess, whereLessEq**.

Принимает key и val1 и составляет объект сравнения с соответствующим названию типом (=, !=, >, >=, <, <=).

4.2.1.2.2.2.3.3. **whereBetween**. Принимает key, val1 и val2, и составляет объект сравнения с ними. В конечном итоге, на этапе формирование sql этот объект будет преобразован в два условия с >= и <=.

4.2.1.2.2.2.3.4. **whereIn, whereNotIn**. Принимает key и val1. В классическом сценарии использование val1 это массив, но для удобства использования, если в val1 передано одно значение, то метод перенаправит в whereEq. Если же передан все же массив, то на этапе формирования sql он будет преобразован в where <field> in (...val1). Разумеется значения будут заэскейпены.

4.2.1.2.2.2.3.5. **whereInSelect, whereNotInSelect**. Принимает key и val1. val1 это строка sql вида “select id from <> where …“. На этапе формирования sql будет сформировано условия вида where id in (select id from … where…). Такая строка может быть сформирована другим запросом, с параметром prepareSQL: true. Напомню базовую концепцию, что мы очень редко пишем sql код сами, поэтому этот вид сравнения, в большинстве случаев используется с **prepareSQL**.

4.2.1.2.2.2.3.5.1. Этот тип сравнения блокируется, если запрос пришел не из внутреннего метода, а с

клиента (фронт или API), чтобы исключить возможность свободно подставить sql код.

4.2.1.2.2.2.3.6. **whereLike, whereStartLike, whereEndLike**. Методы для сравнения вида LIKE. whereLike формирует тип сравнения '%val1%', whereStartLike - 'val1%' (начинается с…), whereEndLike

4.2.1.2.2.2.3.7. Группы. groupOr, groupAnd.

4.2.1.2.2.2.3.7.1. Массив условий, по умолчанию, это группа условий соединенных через AND, что равносильно groupAnd, однако, если вам нужна группа условий через OR, для этого используется groupOr. Функция позволяет отделить группу скобками, а все условия внутри будут идти через OR.

4.2.1.2.2.2.3.7.2. Например, “все пользователи, у которых gender=Male AND (Name=Alex OR Nickname=Alex)”, будет выглядеть так:

```
p = {
   where:[
      whereEq('gender_sysname', 'MALE'),
       groupOr([
           whereEq('firstname', 'Alex'),
           whereEq('nickname', 'Alex')
       ])
   ]
}
res = await r.api(User, 'get', prepareP(p))
if (res.code) return res

const users = res.data.rows
```

4.2.1.2.2.2.3.7.3. Внутри группы, также может быть группа. Пример:

```
where:[
```

```
  whereEq('gender_sysname', 'MALE'),
   groupOr([
       whereEq('firstname', 'Alex'),
       groupAnd([
           whereEq('use_nickname', true),
           whereEq('nickname', 'Alex'),

       ])

   ])
]
```

4.2.1.2.2.2.3.7.4. .

4.2.1.2.2.2.4. param_where

4.2.1.2.2.2.4.1. Устаревшая форма, которая сокращала запись в прошлой версии ядра однако в данной версии не актуально, так как появились удобные функции обертки. Это объект {[field: string]:any}, где field это имя поля, а правая часть - значение. Тип сравнения для param_where всегда “=”.

4.2.1.2.2.3. limit | page_no | offset

```
limit?: false | number // false - no limits
page_no?: number
offset?: number
```

4.2.1.2.2.3.1. Накладывает LIMIT в SQL запрос SELECT. По умолчанию равен 10000. Сделано так, чтобы без необходимости не получать всю таблицу.

4.2.1.2.2.3.2. Можно передать false, тогда LIMIT не будет добавлен, или число. Если попытаться передать что-то отличное от указанных выше, например sql инъекцию, параметр будет проигнорирован и будет применен limit по умолчанию.

4.2.1.2.2.3.3. Можно задать смещение с помощью offset или page_no. По умолчанию 0. Если передан page_no (но не передан offset), смещение будет высчитано как page_no * limit.

4.2.1.2.2.4. sort

```
sort?: {
   columns: string[]
   directions?: string[]
```

```
} | string | false
```

4.2.1.2.2.4.1. Определяет сортировку. По умолчанию сортировка определяется профайлом Класса, и, как правило, это “id DESC” (новые записи вверху).

4.2.1.2.2.5. prepareSQL

```
prepareSQL?: boolean
```

4.2.1.2.2.5.1. Описывалось в разделе про **where->whereInSelect**. Если параметр указан как true, то результирующий SQL запрос будет только собран, но вместо выполнения будет возвращен строкой в res.data.sql. Это можно сказать queryBuilder, который соберет запрос, учитывая все особенности ядра, автоматические join, проверки, типы полей, настройки профайла. Акцентирую внимание еще раз, что мы крайне редко пишем делаем прямые SQL запросы, составленные вручную или с помощью данного параметра, поэтому результат выполнения данного запроса следует использовать в последующих запросах посредствам базового API, то есть в whereInSelect. В дальнейшем могут появляться и другие конструкции, чтобы использовать этот код не только в where, но и например в union или from subquery, но только по мере необходимости.

4.2.1.2.2.5.1.1. Чтобы было яснее, вот пример запроса:

```
p = {
   columns:['id'],
   where: [
       whereEq('status_sysname', 'ACTIVE'),
       groupOr([
           whereEq('firstname', 'Alex'),
           groupAnd([
               whereEq('use_nickname', true),
               whereEq('nickname', 'Alex'),
           ])
       ])
   ],
   prepareSQL:true,
}
res = await r.api(User, 'get', prepareP(p))
if (res.code) return res
```

```
const alexUsersSql = res.data.sql

p = {
   columns: ['id', 'name'],
   where: [
       whereEq('is_active', true),
       whereInSelect('customer_user_id', alexUsersSql),
   ]
}
res = await r.api(Deal, 'get', prepareP(p))
if (res.code) return res
const deals = res.data.rows
```

Результат:

```
SELECT
   `deal`.`id`,
   `deal`.`name`,
   `deal`.`customer_user_id`,
   `deal`.`is_active`
FROM
   `deal`
WHERE
   (
      `deal`.`is_active` = TRUE
      AND `deal`.`customer_user_id` IN
      (
           SELECT
               `user`.`id`
           FROM
               `user`
               LEFT JOIN `user_status` AS `user_status1` ON (
                   `user`.`status_id` = `user_status1`.`id`
               AND `user_status1`.deleted IS NULL)
           WHERE
               (
                   `user_status1`.`sysname` = 'ACTIVE'
                   AND (
                       `user`.`firstname` = 'Alex'
                       OR ( `user`.`use_nickname` = TRUE AND `user`.`nickname` = 'Alex' )
                   )
               )
               AND `user`.deleted IS NULL
      )
   )
   AND `deal`.deleted IS NULL
ORDER BY `deal`.`id` DESC LIMIT 0, 10000
```

4.2.1.2.2.6. countOnly

4.2.1.2.2.6.1. Не делает запрос непосредственно данных, но делает запрос count, по заданным условиям (игнорируя limit). Возвращает результат в res.data.count.

4.2.1.2.2.7. enableInheritValues: boolean

4.2.1.2.2.7.1. Для иерархических сущностей, ссылающихся самих на себя, есть возможность получать родительские значения, если нет своего. Это работает для полей, у которых в профайле установлен параметр “is_inherit”. Система при запросе get, получит ближайшего родителя со значением (в наследуемом поле) для каждой записи, у которой нет своего

значения. В качестве значения будет подставлено не само значение, а объект интерфейса **IInheritedValue**, содержащий само значение и информацию о родителе, от которого удалось отнаследовать. Клиентские компоненты, такие как формы, фреймы и таблицы умеют работать с такими значениями и корректно отображают данные.

4.2.1.2.2.8. enableMultiValues: boolean

4.2.1.2.2.8.1. Когда запрос делается к классу (клиентскому объекту), с динамическими полями, то для одной записи может быть несколько значений. Эти значения хранятся в отдельной таблице и имеют связь один ко многим. Система умеет самостоятельно собирать все значения b возвращать их в виде строки (enableMultiValues=false) разделяя значения сепаратором “-|-” (сепаратор может быть переопределен на уровне класса или запроса: multi_value_separator), или вернуть объектом (enableMultiValues=true) вида {values, valueColumnName}. По умолчанию enableMultiValues = false.

4.2.1.2.2.8.2. Этот параметр может быть определен на уровне запроса или профайлом класса/клиентского объекта.

4.2.1.2.2.9. co_path___

4.2.1.2.2.9.1. Передается с клиента автоматически, когда запрос (get и getProfile и другие базовые методы) отправляется из форм, фреймов или таблиц. Используется ядром системы для механизма динамических полей, когда системе надо знать, что мы запрашиваем данные сущности, например таблицы, расположенной внутри родительской формы

(родительской записи), которая может ограничивать не только набор данных, но и набор полей (как самой этой сущности, так и полей из других сущностей). Примером таких динамических полей, могут служить, например, характеристики товаров, которые в зависимости от категории, могут быть разные. Так, для телевизоров это может быть диагональ экрана и кол-во пикселей на дюйм, а для плит, количество конфорок. Так как такие характеристики это не физические поля в таблице товаров, а данные в отдельных таблицах, то ядру необходимо понимать, какие поля и из каких таблиц необходимо подгрузить в зависимости от категории товара, так чтобы пользователь мог работать с этими данными в таблице, как с обычными полями.

4.2.1.2.2.10. group_by

4.2.1.2.2.10.1. Позволяет определить группировку по заданным полям. Может быть строкой, тогда это воспринимается как имя поля, или массивом строк, тогда это перечисление полей.

4.2.1.2.2.10.2. Как правило используется с параметром **specColumns** (см. ниже).

4.2.1.2.2.11. specColumns

4.2.1.2.2.11.1. Позволяет указать особые поля, применить функции агрегации или другие SQL функции.

4.2.1.2.2.11.2. Объект, где ключ альяс результирующего столбца. А значение - SQL выражение.

```
specColumns: {
   total: 'sum(amount)',
},
```

4.2.1.2.2.11.3. Обычно используется с groupBy.

4.2.1.2.2.11.4. Для обеспечения безопасности, параметр запрещен для запросов с клиента.

4.2.1.2.3. Выходные параметры. Интерфейс IAPIResponse/IError

4.2.1.2.3.1. Интерфейс IAPIResponse соответствует IError, так что вы можете встретить оба варианта. Используйте IAPIResponse.

4.2.1.2.3.2. В случае ошибки объект будет инстансом класса MyError или UserError.

4.2.1.2.3.3. Успешный ответ будет содержать информацию в объекте data:

4.2.1.2.3.3.1. rows. Массив объектов, где каждый объект это запись, в общем случае соответствующая интерфейсу IAPIResponseDataRow, но в частном расширенному интерфейсу того класса, к которому был запрос, например IDealDataRow (такие интерфейсы создаются системой автоматически (/classes/_structures/<class_name>.ts) при создании класса и содержат все его поля).

4.2.1.2.3.3.2. additionalData. Объект, содержит дополнительную информацию.

4.2.1.2.3.3.2.1. count. Сколько записей вернулось по заданному запросу, с учетом limit.

4.2.1.2.3.3.2.2. count_all. Сколько всего записей по заданным условиям, без учета limit.

4.2.1.2.3.3.2.3. data_columns. Массив имен полей (строк), в последовательности определенной профайлом. Используется системой для оптимизации данных при передаче.

4.2.1.2.3.3.2.4. additional_class_fields_profile. Содержит дополнительную информацию о наследуемых полях, которая используется клиентскими компонентами.

4.2.1.2.3.3.2.5. На сервере, до отдачи клиенту (если метод это предполагает), можно увидеть в объекте и другие данные (на клиент они не передаются):

4.2.1.2.3.3.2.5.1. cFProfile. Объект с профайлами каждого поля запрашиваемого класса. Это также используется системой в некоторых случаях.

4.2.1.3. add

4.2.1.3.1. Создаст новую запись в таблице класса, сбросит связанный кэш, сохранит историю (см. Механизм истории). Создание может быть запрещено настройками профайла (new_command|editable на уровне класса/клиентского объекта).

4.2.1.3.2. Входными параметры. Интерфейс IAPIQueryParams

4.2.1.3.2.1. Те поля которые вы хотите заполнить при создании.

4.2.1.3.2.1.1. Поля запрещенные для добавления в профайле (editable/server_editable/insertable/server_insertable) будут проигнорированы.

4.2.1.3.2.2. Вы можете указывать и виртуальные поля, такие как sysname, тогда система сама отправится в соответствующий справочник, найдет id этого значения (чаще всего из кэша), подставит его в соответствующее физическое поле.

4.2.1.3.2.2.1. Применимо для виртуальных полей ссылающихся на справочники.

4.2.1.3.2.3. Для некоторых полей может быть указано значение по умолчанию в настройках

профайла. Оно будет подставлено, если само значение не передано.

4.2.1.3.2.3.1. Значение может быть указано для виртуального поля. Например, для Заказа, у которого есть поля status_id, status и status_sysname, может быть указано значение по умолчанию для поля status_sysname="CREATED". Система найдет значение в справочнике и подставит его в status_id (конечно если status_id или status_sysname не переданы при создании).

4.2.1.3.2.4. Система проверит записи на уникальность, согласно настройкам профайла класса/клиентского объекта.

4.2.1.3.2.4.1. В случае обнаружения неуникальности система вернет ошибку (UserError) recExist (code:105).

4.2.1.3.3. Выходные параметры. Интерфейс IAPIResponse/IError

4.2.1.3.3.1. В случае ошибки метод вернет объект ошибки - инстанс UserError/MyError.

4.2.1.3.3.2. В случае успеха метод вернет id добавленной записи в объекте data инстанса UserOk (res.data.id).

4.2.1.4. modify

4.2.1.4.1. Изменит указанную запись в таблице класса, сбросит связанный кэш, сохранит историю. Изменение может быть запрещено настройками профайла (modify_command|editable на уровне класса/клиентского объекта).

4.2.1.4.1.1. Как указано в описании выше, метод изменяет именно конкретную запись. Система не поддерживает массовое изменение по условиям. Это сделано специально, так как так выше контроль со стороны разработчика и логирования за изменениями. Пока не было задач, для которых потребовалось бы настолько массово провести изменения за короткий

срок, что изменение по одной записи (можно в параллели) не подходило бы.

4.2.1.4.1.1.1. При необходимости такой метод может быть разработан и возможно появится в следующих версиях ядра.

4.2.1.4.2. Входными параметры. Интерфейс IAPIQueryParams

4.2.1.4.2.1. id изменяемой записи и те поля, которые вы хотите изменить.

4.2.1.4.2.1.1. Поля запрещенные для изменения в профайле (editable/server_editable/updateable/server_updateable) будут проигнорированы.

4.2.1.4.2.2. Вы можете указывать и виртуальные поля, такие как sysname, тогда система сама отправится в соответствующий справочник, найдет id этого значения (чаще всего из кэша), подставит его в соответствующее физическое поле.

4.2.1.4.2.2.1. Применимо для виртуальных полей ссылающихся на справочники.

4.2.1.4.2.3. Система проверит записи на уникальность, согласно настройкам профайла класса/клиентского объекта.

4.2.1.4.2.3.1. В случае обнаружения неуникальности система вернет ошибку (UserError) recExist (code:105).

4.2.1.4.3. Выходные параметры. Интерфейс IAPIResponse/IError

4.2.1.4.3.1. В случае ошибки метод вернет объект ошибки - инстанс UserError/MyError.

4.2.1.4.3.2. В случае успеха метод вернет id измененной записи в объекте data инстанса UserOk (res.data.id).

4.2.1.5. remove

4.2.1.5.1. Удаление в системе работает каскадно. Система определяет все зависимые классы (таблицы) и тип их зависимости. После чего по этим классам проводится поиск, есть ли зависимые данные от удаляемой записи.

4.2.1.5.1.1. Зависимость может быть родитель-ребенок, тогда при удалении записи будут удалены все зависимые от нее записи, или зависимость вида использования этой записи в других сущностях не как родительской, а как справочной, тогда такие эти поля вычищаются.

4.2.1.5.1.2. Зависимость учитывает иерархические таблицы и для всех зависимых данных ищутся их зависимости.

4.2.1.5.1.3. Такое удаление требует подтверждения, для этого должен быть передан параметр **confirm**=true. Если параметр не передан, то после поиска зависимостей система возвращает UserError типа needConfirm (code=10), а в data передает объект с информацией по всем зависимостям.

4.2.1.5.1.3.1. Код 10 особым образом обрабатывается фронтом автоматически показывая диалог с запросом подтверждения, с указанным на бэкэнде текстом и верстkой. Вы можете использовать needConfirm в своих пользовательских методах.

4.2.1.5.2. Удаление может быть запрещено на уровне профайла класса/клиентского объекта (editable/remove_command).

4.2.1.5.3. Система использует механизм soft deletion, то есть данные не удаляются, а помечаются меткой времени. Ядро строит все запросы таким образом, что везде (включая join(ы)) подставляется условие проверки поля **deleted**.

4.2.1.5.4. Входными параметры. Интерфейс IAPIQueryParams

4.2.1.5.4.1. id удаляемой записи и булевое поле confirm=true.

4.2.1.5.5. Выходные параметры. Интерфейс IAPIResponse/IError

4.2.1.5.5.1. В случае ошибки метод вернет объект ошибки - инстанс UserError/MyError.

4.2.1.5.5.2. В случае успеха метод вернет id удаленной записи в объекте data инстанса UserOk (res.data.id).

4.2.2. before/after

4.2.2.1. Для операций add/modify/remove предусмотрены методы для выполнения действий до и после выполнения самого метода. Для использования механизма в классе нужно определить соответствующий метод с классическими входными и выходными параметрами (IAPIRequest и IAPIResponse).

4.2.2.1.1. before. Для каждого метода проверяется наличие соответствующего метода в классе. Для add - beforeAdd, для modify - beforeModify, для remove - beforeRemove.

4.2.2.1.1.1. В вызываемые методы передаются параметры переданные в основной метод.

4.2.2.1.2. after. Аналогично, для каждого метода проверяется наличие соответствующего метода в классе. Для add - afterAdd, для modify - afterModify, для remove - afterRemove.

4.2.2.1.2.1. В вызываемые методы передается объект с id (для add это будет id созданной записи), origParams (параметры переданные в основной метод) и системный rollback_key.

4.2.2.1.3. Также можно определить общий метод, просто ”before”/“after” - он будет срабатывать (при наличии), до/после любого из действий (add/modify/remove).

4.2.3. Прочие get

4.2.3.1. getById

4.2.3.1.1. Упрощенный метод для получения записи именно по id. Он не принимает where и param_where в параметрах, а вместо этого получает id. Также как и базовый get он принимает columns и прочие, применимые и осмысленные к параметры.

4.2.3.1.2. В отличии от get, в случае если такая запись не будет найдена, то это вызовет ошибку “Запись не найдена”.

4.2.3.1.3. Найденная запись будет располагаться как и при обычном get в res.data.rows под нулевым индексом, то есть в res.data.rows[0]. В последней, на момент написания версии ядра, эта запись также помещена в res.data.row.

4.2.3.2. getCount

4.2.3.2.1. Вызывает метод get, с параметром **countOnly=true** (см. описание get), и соответственно возвращает количество записей в res.data.count

4.2.3.3. getForSelect

4.2.3.3.1. Метод используется для выпадающих списков. Пока не описан подробно

4.2.3.4. getForFilterSelect

4.2.3.4.1. Метод используется для выпадающих списков для фильтров. Пока не описан

4.2.3.5. search

4.2.3.5.1. Метод получает строку и подставляет ее в where (с экранированием) для каждого поля, отмеченного в профайле как quick_search_field.

4.2.3.6. getMass

4.2.3.6.1. Deprecated.

4.2.3.7. export_to_excel

4.2.3.7.1. Формирует excel с теми же настройками и данными, что и сама клиентская таблица из которой вызывается.

4.2.3.8. export_to_json

4.2.3.8.1. Формирует json с теми же настройками и данными, что и сама клиентская таблица из которой вызывается.

4.2.4. System

4.2.4.1. clearCache

4.2.4.2. getDependence

4.2.5. Иерархические

4.2.5.1. getParentIds

4.2.5.2. getParents

4.2.5.3. getChildIds

4.2.5.4. getChildren

4.2.5.5. fixNodeDeep

4.2.5.6. getTree

4.2.5.7. getTreeChilds

4.2.6. History

4.2.6.1. getRecHistory

4.2.6.2.

4.2.7. Административные методы

4.2.7.1. unlockRec

4.2.7.2. showLocks

4.2.8. Прочие

4.2.8.1. getDefault

4.2.8.2. setDefault

4.2.8.3. setColumnPosition

4.2.8.3.1. Применим только для полей классов или клиентских объектов

4.2.9. Подписки (экспериментальное)

4.2.9.1. subscribe

4.2.9.2. unsubscribe

4.2.9.3. unsubscribeAll

4.2.10.

5. Класс Table

5.1. Это системный класс. В нем реализована логика синхронизаций tables.json с базой данных.

6. Прочие Классы и методы ядра

6.1. User|User_session

6.1.1.

6.1.2. get_me/getMe

6.1.3. getConfig

6.1.4. getRoles

6.1.5. Авторизация, сверка пароля, регистрация, восстановление пароля, сделать активным.

6.1.5.1. login

6.1.6. Механизм pinCode

6.1.7. updateRoleList

6.1.8. Режимы пользователей

6.1.8.1. См. описание механизма

6.1.9. removeSelf

6.1.10. sendMsg

6.1.11. uploadAndSetAvatar/clearAvatar

6.1.12.

6.2. Origin

6.3. Support

6.3.1. sendSupport

6.4. ClassTemplate

7. Профайл класса/клиентского объекта и их полей.

7.1. Описание профайла самого класса/КО

7.1.1. Описание

7.1.1.1. Настройки профайла позволяют сформировать поведение системы при взаимодействии с ним. Это настройки

относящиеся как к серверной стороне, так и настройки для клиентских компонентов.

7.1.2. Поля профайла самого класса.

7.1.2.1. name. Должно совпадать с ключом объекта, описывающего этот класс.

7.1.2.2. name_ru. Не обязательно должно быть на русском. Это титул, который будет отображаться в клиентских таблицах, а также использоваться в системных сообщениях при манипуляциях с этим объектом, например “Пользователь добавлен”.

7.1.2.3. primary_key. Системное, всегда “id”.

7.1.2.4. parent_table и parent_key. Используется клиентскими компонентами, таблицами расположенными внутри формы или фрейма. Позволяет им находить родительскую форму, получать id родительской записи (той, для которой открыта форма) и накладывать фильтр на данные таблицы.

7.1.2.4.1. parent_key используется или этот или берется из профайлов полей - поле у которого стоит галочка parent_key.

7.1.2.5. server_parent_table и server_parent_key. Определяет зависимость таблицы (прямую иерархию) от других. Может быть указано несколько через запятую. Количество ключей должно совпадать с количеством таблиц. Допускается зависимость от себя самой, например таблица d_location или d_storage - Внутри страны могут быть области, а внутри Склада, стеллажи.

7.1.2.5.1. Используется для определения зависимостей при каскадном удалении и очистке кэша.

7.1.2.6. server_parent_key_for_dynamic_fields. Не используется.

7.1.2.7. checkbox_where, checkbox_where_default_enabled и checkbox_where_title. Позволяет вывести в заголовке таблицы чекбокс, при активации которого на данные накладывается дополнительное условие указанное в checkbox_where в виде массива объектов условий, преобразованного в строку с помощью JSON.stringify. Объект условий такой же как и на сервере, соответствует интерфейсу IOneWhere. Самая простая форма этого объекта содержит поля только key и val1.

7.1.2.7.1. Для текущей даты предусмотрена подстановка, то есть если в val1 написать NOW_DATE, то она будет заменена на дату в формате “DD.MM.YYYY”. Это было нужно в одном из проектов и может быть расширено или урезано, при необходимости, в tableNew.js

7.1.2.7.2. Как понятно из названий двух дополнительных параметров, можно задать начальное состояние чекбокса и его лейбл.

7.1.2.8. param_checkbox и его дополнительные поля. Не используются.

7.1.2.9. ending. Используется в русском языке для добавления склонения при формировании сообщений при манипуляциях с данными.

7.1.2.9.1. Например, для сущности женского рода, “Локация”, нельзя написать “Локация изменен” (при изменении записи в таблице или форме), а нужно написать “Локация изменен**а**”. Для этого мы пишем букву “а” в поле ending и система начинает формировать сообщения правильно.

7.1.2.10. default_order_by. Позволяет указать сортировку по умолчанию при get запросах. Если при создании класса ничего не указать, то там по умолчанию будет "id desc", то есть показывать самые свежие записи в самом верху.

7.1.2.11. default_where. Для класса можно наложить условия по умолчанию. Также как описано для поля “checkbox_where”, это строка JSON, с массивом условий в стандартном для “where” виде (IOneWhere[]).

7.1.2.11.1. Эти условия будут накладываться при любом запросе.

7.1.2.11.1.1. Если запрос делать не с клиента, а из метода на сервере, то можно отменить наложение передав параметр doNotUseDefaultWhere=true в параметрах запроса (rParams).

7.1.2.12. open_form_client_object. Используется в клиентском компоненте таблиц. Позволяет указать, какую форму открыть из контекстного меню. Имя формы, это имя клиентского объекта, который должен быть создан в системе.

7.1.2.12.1. Также должна стоять галочка new_by_modal_command.

7.1.2.13. child_client_object. Затрудняюсь утверждать, что этот параметр используется.

7.1.2.14. rows_max_num_list и rows_max_num. Позволяет указать, какие именно варианты “по сколько строк выводить в таблице” будут в выпадающем списке клиентского компонента таблицы. По умолчанию это "10,20,50,100", но вы можете переопределить это для какого-нибудь класса.

7.1.2.14.1. rows_max_num. Позволяет указать начальное состояние, по сколько выводить.

7.1.2.14.2. Следует отметить, что выбор сохраняется для пользователя.

7.1.2.15. new_command, modify_command, remove_command. Это чекбоксы. Определяют можно ли редактировать этот данные этого класса. Соответственно Создавать, Изменять, Удалять. Для таблиц, при отсутствии галочки, соответствующие кнопки не будут выводиться.

7.1.2.15.1. Для запросов с клиента состояние этих галочек проверяется и на стороне сервера при соответствующих командах.

7.1.2.15.1.1. *появилось (серверная проверка) недавно относительно написания этой документации и в некоторых проектах может отсутствовать.

7.1.2.16. editable. Если false, то запрещает любые действия с данными со стороны клиента. Проверяется на сервере.

7.1.2.17. additional_functionality. Используется только для клиентского компонента таблиц. Если установлена, то будет подключаться клиентский js файл, одноименный с клиентским объектом. Файл должен располагаться в той же директории что и верстка.

7.1.2.18. distinct_columns. Если указать строкой имена полей через запятую, то в запрос будет добавляться DISTINCT по этим полям.

7.1.2.18.1. Поля естественно не будут подставлены как есть, то есть здесь не может быть SQL инъекции.

7.1.2.19. use_cache. Можно определить, что для этого класса не должен использоваться механизм кэширования. Это нужно только для системных таблиц, например Сессий, хотя и для них не обязателен.

7.1.2.19.1. Имеется такой же параметр, который можно использовать в запросах на стороне сервера, но он также редко нужен, в основном в системных механизмах. Например в механизме миграции данных, где для ускорения работы мы не сбрасываем кэш для каждой добавляемой/изменяемой строки, а делаем это в конце пачки, однако в процессе нам может потребоваться запросить актуальные данные (не из кэша).

7.1.2.20. check_company_access. Так и не реализованный функционал разграничения доступа к данным по “компаниям”. Вместо этого есть другой механизм ограничения доступа к данным.

7.1.2.21. count_large. Для больших таблиц (на практике оказалось не нужно и не эффективно). Если установлена в true, то запрос на подсчет реализуется иначе. Устарело.

7.1.2.22. like_type. Определяет для класса какой именно режим LIKE будет использоваться для быстрого поиска в таблицах, выпадающих списках полей этого класса и фильтрах этого класса. Возможные варианты, как и разные типы сравнения в IOneWhere: “%like” (“заканчивается введенным текстом”), “like” (“в любом месте введенный текст”), “like%” (“начинается с введенного текста”). По умолчанию “like”.

7.1.2.23. hierarchical_table и do_not_set_deep. Галочка определяет что таблица ссылается сама на себя. Система сама определяет и выставляет эту галочку при синхронизации класса. Используется для внутренних механизмов связанных именно с иерархическими таблицами. Также для таких таблиц автоматически добавляются специфические поля, такие как node_deep, parents_count, children_count.

7.1.2.23.1. За состоянием этих полей система следит автоматически. Но это можно отключить настройкой do_not_set_deep.

7.1.2.23.2. Для таких таблиц доступны методы getTree, getParentIds, getChildIds.

7.1.2.24. do_not_use_cache_client. Устарела. Использовалась для того, чтобы на стороне клиента не кэшировался getProfile в LocalStorage. Сейчас кэш профайла не используется.

7.1.2.25. child_class. Позволяет указать дочерний класс и при необходимости клиентский объект. Это используется, если имеется иерархия нескольких сущностей. Например, Строение-Этаж-Помещение - 3 сущности в общей иерархии. Указание этого поля, позволяет методу getTree построить дерево не только одной иерархической сущности, но и нескольких, как в примере. Результатом getTree будет не только набор записей, но и информация об их сущности, что позволит на клиенте корректно их отображать и проводить различные операции.

7.1.2.25.1. Дочерний класс указывается строкой, а при необходимости уточнить его клиентский объект ставится точка и без пробелов пишется имя КО. Например: “Level.table_level”.

7.1.2.26. command_get, command_add, command_modify, command_remove.

7.1.2.26.1. Это поля типа “строка”. Позволяет указать, какую команду отправлять вместо базовых команд (“get”, “add”, “modify”, “remove”). Может использоваться клиентскими компонентами. Однако в текущей версии, только компонент форм реализует использование только параметра “command_get”, а остальные команды заложены на будущее. Это следует доработать и довольно легко сделать при надобности.

7.1.2.26.2. new_by_modal_command. Используется только в клиентском компоненте таблиц и определяет, показывать ли пункт контекстного меню “Открыть в форме”. Пункт будет, при условии, что галочка установлена, а в поле open_form_client_object указано имя формы.

7.1.2.26.3. enable_multi_values. Позволяет выводить специальным объектом несколько значений для одного поля. Подробнее см. параметр “enableMultiValues” метода “get”. Параметр приоритетнее профайла.

7.1.2.26.4. mysql_unique. Включает поддержку защиты уникальности на уровне СУБД. То есть поля с параметром is_unique должны быть уникальны и они создаются в базе именно уникальными. Если параметр выключен, работает только механизм ядра, который при добавлении/изменении делает быстрый запрос на поиск данных с таким же сочетанием.

7.2. Описание профайла полей класса/КО

7.2.1. Описание

7.2.1.1. В отличии от профайла класса, где настройки относятся к самому классу, тут настраивается поведение каждого его поля в отдельности. Сюда попадают во-первых, системные настройки, необходимые для работы ядра (например информация о типе поля и, если поле виртуальное (физически не располагается в этой таблице), то информация о связи с другими таблицами или из каких полей/строк его собрать. А во-вторых, настройки поведения, как для серверной части, так и для клиентских компонентов.

7.2.1.2. В описании будут встречаться виртуальные поля, то есть поля из других таблиц, со своим ключем. Например class (class_id). Мы не будем акцентировать, что физически храниться именно поле <что-то>_id, чтобы не писать везде одно и то же.

7.2.2. Поля

7.2.2.1. class_id. Системное поле, поле связи с классом.

7.2.2.2. column_name. Это системное имя поле и оно не должно изменяться после создания. Именно так поле называется в tables.json и в базе данных.

7.2.2.3. name. Это пользовательское наименование поля (user friendly). Эта надпись будет в таблицах и формах.

7.2.2.4. select_class. Необходим для клиентских компонентов для виртуальных полей (подключенных из других таблиц), чтобы выпадающие списки знали в какую таблицу. Как правило там нужно указать тоже что и в from_table. Если ничего не указано, то ядро может использовать from_table, однако эта опция появилась недавно и не тестировалась полноценно,

так что если выпадающие списки не туда обращаются, просто укажите верный класс в этом поле.

7.2.2.5. parent_key. См. также описание parent_key в профайле самого класса. Позволяет указать, что по этому полю будет применена фильтрация данных в таблице, если эта таблица расположена внутри формы родительской сущности. Например в форме заказа (№15), можно разместить таблицу с позициями заказа и указать этой таблице, что parent_key это “order_id”, тогда будут выведены только позиции этого (15-го) заказа.

7.2.2.6. primary_key. Системное. Всегда id.

7.2.2.7. foreign_disable (foreign_table, foreign_key_str). Ядро может автоматически генерировать внешние ключи для полей ссылающихся на другие таблицы. Этот механизм был введен относительно недавно, и он по умолчанию создавал внешние ключи, но это можно было отключить параметром foreign_disable. Далее стало ясно, что лучше не создавать ключи для всех, а делать это точечно. Поэтому теперь этот параметр по умолчанию имеет значение true.

7.2.2.7.1. foreign_table, foreign_key_str - системные поля, куда вписывается определенные системой зависимости.

7.2.2.8. return_id, return_name, lov_return_to_column, select_search_columns. Поля для настройки редактора вида выпадающий список.

7.2.2.8.1. Запрос направляется в select_class и берет оттуда id и name. Однако это можно переопределить.

7.2.2.8.2. return_id. Можно указать чтобы из запрашиваемой таблицы возвращалось не id, а какое-нибудь другое поле. Это может быть полезно, если сам запрос переопределен и например делает группировку по какому-нибудь полю, тогда именно его имеет смысл брать как id для списка.

7.2.2.8.3. return_name. Это поле переопределяется чаще, чем return_id, так как даже если запрос идет в классический справочник, то не всегда из него нужен именно name. Например из справочника d_cities может потребоваться поле name_full (поле, например, содержит полный адрес города, с указанием страны и области).

7.2.2.8.4. lov_return_to_column. Определяет в какое поле подставить выбранное значение. Если, например, редактор типа “Выпадающий список” указан для поля city, то система по умолчанию, посмотрит, какой keyword у этого поля, то есть через какое поле осуществляется связь со справочником, и поняв что это city_id установит выбранное значение из списка именно туда. Но иногда требуется переопределить это поведение.

7.2.2.8.5. select_search_columns. Позволяет указать через запятую поля, по которым будет осуществляться поиск в справочнике (в выпадающем списке есть поле ввода для фильтрации значений). По умолчанию поиск идет по полю указанному в return_name.

7.2.2.9. select_autocomplete_columns. Используется при работе с дополняемым списком. Часто при заполнении справочника недостаточно просто указать одно поле, а нужно заполнить несколько. В этом случае системе нужно знать, какие еще поля требуется заполнить, чтобы добавление прошло успешно. Можно перечислить эти поля через запятую.

7.2.2.9.1. Я не уверен что этот механизм работает в текущей версии ядра.

7.2.2.10. sort_no. Важный параметр, определяет последовательность колонок в таблице. Менять его можно через контекстное меню в настройках полей класса или клиентского объекта “Set column position".

7.2.2.10.1. Имеется важная особенность, которую следует учитывать при сортировке. Поля, использующие другое поле как ключ связи (относится к виртуальным полям, которые подтягиваются из других таблиц посредствам JOIN, используя ключ указанный в “keyword”), должны располагаться после этого ключа.

7.2.2.10.1.1. Кроме этого, у вас могут быть поля, которые подключены через JOIN через несколько таблиц (у них прописан параметр “join_table”), такие поля должны располагаться после одного из (любого) виртуального поля, использующего ту таблицу, через которую осуществляется соединений. Если

соединение осуществляется через несколько таблиц, то соответственно будет несколько полей, которые должны располагаться перед.

7.2.2.10.1.2. Думаю в дальнейшем ядро будет доработано и порядок JOIN будет определяться автоматически, но пока следует учитывать это ограничение.

7.2.2.11. type, field_length. Системные поля определяющие тип и длину поля так, как это необходимо СУБД. Они заполняются в tables.json и не правятся через интерфейс.

7.2.2.11.1. Если вы создали не с тем типом, то поле следует удалить из tables.json, синхронизировать и создать заново.

7.2.2.12. lov_columns. В настоящее время не используется.

7.2.2.13. filter_type (filter_type_id). Позволяет выбрать, какой тип фильтра должен быть у этого поля. Фильтры появляются в клиентском компоненте таблиц, если есть хотя бы одно поле, у котого выставлен тип фильтра. Если тип фильтра не выбран, то фильтра по этому полю не будет.

7.2.2.13.1. Фильтры выбираются исходя из типа поля. Это может быть диапазон чисел, даты, даты и времени, чекбокс, текст (строгое совпадение), текстLike (совпадение части строки), выпадающий список и выпадающий список с пустым значением.

7.2.2.13.1.1. Выпадающий список работает на основе существующих данных, то есть там будут только те уникальные значения, которые встречаются в данных этой таблицы.

7.2.2.13.1.1.1. Есть желание сделать возможность переключения режима, чтобы для полей относящихся к справочникам, например Статус, запрос был именно к справочнику, а не запрос данных с группировкой по этому полю. Но это не редко нужно.

7.2.2.13.1.1.2. Кроме этого в одном проекте, еще на первой версии ядра, эти фильтры доработаны (по мере необходимости будет перенесено):

7.2.2.13.1.1.2.1. Список предоставляет не только уникальные значение но и подсчет количества вхождений, например Статус: Создан(5), В работе (8)

7.2.2.13.1.1.2.2. А также есть возможность указать фильтр опираясь на данные в отдельной таблице со связью один ко многим, например виртуальное поле Исполнители, может обращаться к таблице ИсполнителиЗаявки и выдавать уникальные значения оттуда.

7.2.2.13.1.1.2.3. Мультиселект. Позволяет выбрать несколько значений.

7.2.2.14. type_of_editor (type_of_editor_id). Схож с типом фильтров, но определяет какой редактор будет у этого поля для редактирования/создания. Используется в клиентских компонентах (формы, фреймы, таблицы). Может быть текстом, wysiwyg, textarea, числом, выпадающим списком, если это виртуальное поле подтягивающиеся из другой таблицы, чекбокс, дата, дата и время дни недели (не уверен что работает корректно), изображение (позволяет выбрать изображение с предпросмотром), файл (позволяет выбрать файл), телефон (не уверен что работает корректно), иконка (не знаю как работает).

7.2.2.14.1. Имеется возможность написать новые редакторы, а также можно на уровне конкретного клиентского объекта (его js файла) переопределить поведение и написать вообще что-нибудь кастомное.

7.2.2.15. quick_search_field. Галочка определяет, что по этому полю будет производится поиск при вводе текста в строку быстрого поиска в клиентском компоненте таблиц или при вызове метода search. По умолчанию установлена у поля id.

7.2.2.16. visible. Определяют видимость полей для клиентских компонентов. Если visible=false, то это, во-первых, будет отражено в профайле, а во-вторых, это поле не будет возвращено методом get.

7.2.2.16.1. Важно понимать, что это не влияет на операции add/modify, для них есть отдельные галочки.

7.2.2.17. required. Определяет что это поле обязательно для заполнения при добавлении записи. Это проверяется на бэкенде, а также клиентские компоненты форм, фреймов и таблиц подсвечивают эти поля звездочкой и при попытке сохранить без заполнения.

7.2.2.18. editable. По умолчанию = true. Определяет можно ли это поле редактировать (указывать при создании записи или изменять после). Проверяется на бэкенде, а клиентские компоненты выводят их либо с редактором, либо без.

7.2.2.18.1. Галочка определяет запрет (при editable=false) на редактирование для запросов с клиента, таким образом вы можете изменить это поле из других методов на стороне бэкенда. Это частая практика, когда поле нельзя изменять напрямую, но есть отдельный метод (например с ограниченным доступом) приводящим к его изменению, возможно с какими-нибудь проверками или сайд эффектами.

7.2.2.18.2. Для ограничения редактирования, в том числе, на стороне бэкенда существует галочка server_editable.

7.2.2.19. server_editable. Как указано выше (описания поля “editable”), ограничивает доступ редактирования (add/modify) для запросов из методов бэкэнда, а не с клиента.

7.2.2.20. insertable. По умолчанию = false. Галочка аналогична “editable”, но распространяется только на добавление. Чтобы запретить добавление, нужно чтобы обе галочки были false.

7.2.2.20.1. По аналогии с “editable”, распространяется на запросы с клиента.

7.2.2.21. server_insertable. Как указано выше (описания поля “insertable”), ограничивает доступ добавления для запросов из методов бэкэнда, а не с клиента.

7.2.2.22. updatable. По умолчанию = false. Галочка аналогична “editable”, но распространяется только на изменение. Чтобы запретить изменение, нужно чтобы обе галочки были false.

7.2.2.22.1. По аналогии с “editable”, распространяется на запросы с клиента.

7.2.2.23. server_updatable. Как указано выше (описания поля “updatable”), ограничивает доступ изменение для запросов из методов бэкэнда, а не с клиента.

7.2.2.24. queryable. Поля ограниченные этим параметром (queryable=false) не возвращаются в запросе getProfile и не возвращаются на клиент в методе “get”, однако, возвращаются в “get” в запросах из других методов на бэкенде.

7.2.2.24.1. На практике это используется только во внутреннем механизме динамических полей.

7.2.2.25. whereable. Позволяет запретить применение условий выборки по конкретному полю.

7.2.2.26. from_table, keyword, return_column. Используются только для полей is_virtual=true. Позволяют указать из какой таблицы, по какому ключу и какое поле вернуть, для полей подтягиваемых из других таблиц. Запрос формируется ядром с помощью JOIN. Заполняются в tables.json и не должны редактироваться через интерфейс.

7.2.2.26.1. Для соединений через несколько таблиц существует поле “join_table”, см. ниже.

7.2.2.26.2. keyword с подстановкой. Вроде сейчас не используется.

7.2.2.27. join_table. Позволяет сделать соединение через одну или несколько таблиц. В этом поле указывается, через какую таблицу соединить, при этом в from_table, как и в обычном случае, указывается какую таблицу надо присоединить.

7.2.2.27.1. Важно чтобы указанная в join_table таблица, была ранее подключена в другом поле. То есть сперва у вас идет поле без join_table, а просто с from_table/keyword/return_column, а уже после может быть поле с join_table/from_table/keyword/return_column, где join_table будет равно from_table предыдущего поля.

7.2.2.27.2. Если соединение идет через несколько таблиц, то каждая предыдущая должна быть предварительно также представлена.

7.2.2.27.3. Рассмотрим на примере. Таблица заказа, у него есть поле “пользователь”, из таблицы пользователей, и у

пользователя есть поле “gender”, из таблицы гендеров. Мы хотим, чтобы в таблице заказов были все эти поля:

7.2.2.27.3.1. Мы добавляем user_id (физическое поле для связи)

7.2.2.27.3.2. Добавляем поле user_nickname, чтобы видеть кто это. is_virtual=true; from_table=”user”; keyword=”user_id”; return_column=”nickname”.

7.2.2.27.3.3. Добавляем поле user_gender. is_virtual=true; join_table=”user”; from_table=”gender”; keyword=”gender_id”; return_column=”name”.

7.2.2.27.3.3.1. Поле keyword для поля подключеного через одну или несколько таблиц указывается, как ключ связи, в предпоследней таблице, то есть в той, которая указана в join_table. На данном примере мы видим, что мы присоединяем таблицу “gender”, из таблицы “user”, а в ней именно “gender_id” позволяет соединиться с таблицей “gender”.

7.2.2.28. join_table_alias, join_table_by_alias. Используется, когда внешняя таблица подключается более одного раза для двух разных полей. join_table_alias используется для указания альяса подключаемой таблицы (указанной в from_table), а join_table_by_alias указывается для следующей подключаемой таблице, сообщая что для соединения нужно использовать именно “тот JOIN”, у которого альяс равен указанному.

7.2.2.28.1. Например в таблице order два поля supplier_id и customer_id, а я хочу подтянуть также nickname и пол для каждого. У меня будет:

7.2.2.28.1.1. Для supplier

7.2.2.28.1.1.1. Поле supplier_id, физическое.

7.2.2.28.1.1.2. Поле supplier_nickname. is_virtual=true; from_table=”user”; keyword=”supplier_id”; return_column=”nickname”. А также зададим ему алиас: join_table_alias=”user_supplier”.

7.2.2.28.1.1.3. Поле supplier_gender. is_virtual=true; join_table=”user”; from_table=”gender”; keyword=”gender_id”; return_column=”name”. А также зададим ему через какое именно соединение (так как будет два “join user”) нужно подключать эту таблицу: join_table_by_alias=”user_supplier”.

7.2.2.28.1.2. Теперь тоже самое для customer.

7.2.2.28.1.2.1. Поле customer_id, физическое.

7.2.2.28.1.2.2. Поле customer_nickname. is_virtual=true; from_table=”user”; keyword=”customer_id”; return_column=”nickname”. А также зададим ему алиас: join_table_alias=”user_customer”.

7.2.2.28.1.2.3. Поле customer_gender. is_virtual=true; join_table=”user”; from_table=”gender”; keyword=”gender_id”; return_column=”name”. А также зададим ему через какое именно соединение (так как у нас два “join user”) нужно подключать эту таблицу: join_table_by_alias=”user_customer”.

7.2.2.29. table_alias. Не используйте table_alias, это системное поле.

7.2.2.30. dynamic_field_id. Системное поле используется для динамических полей.

7.2.2.31. modify_in_ext_tbl, modify_in_ext_tbl_key. Используется для механизма динамических полей. Позволяет реализовать редактор данных хранящихся в другой таблице. Механизм, в принципе, можно использовать не только для динамических полей, но только если это понадобится и это потребует вникнуть в реализацию.

7.2.2.32. is_virtual. Всегда указывается для полей, которые подтягиваются из других таблиц (см. from_table/keyword/return_column), но может также использоваться и для других видов виртуальных полей, таких как CONCAT (см. concat_fields).

7.2.2.33. concat_fields. Позволяет указать через запятую или через пробел, что нужно сконкатинировать. Если система обнаружит хоть одну запятую, то она будет считать, что разделять надо по запятым, если нет, то по пробелу. Лучше используйте запятые.

7.2.2.33.1. Каждый элемент может быть просто строка или имя поля. Точнее так, если есть поле с таким именем, то будет подставлено оно, иначе будет подставлено как есть (конечно с экранированием). Если нужно указать пробел, то он тоже должен быть между запятыми. Например fio:

```
"fio" : {"type": "varchar", "length": "255", "concat_fields": "lastname, ,firstname,
,midname", "is_virtual": true, "name": "ФИО"},
```

7.2.2.33.2. А, например, для поля password мы просто выводим “*”:

```
"password": {"type": "varchar","length": "255", "concat_fields": "*", "name": "Пароль",
"is_virtual": true},
```

7.2.2.33.3. extraSQL. В поле concat_fields можно указать “extraSQL”. Тогда система позволит подставлять в это поле свой sql, однако он должен быть определен на этапе init метода класса. То есть его следует переопределить для класса и вписать туда extraSQL в “profile” соответствующего поля, а также указать от каких полей зависит этот sql (dependCols) чтобы система могла собирать необходимые join и корректно работать с кэш.

7.2.2.33.3.1. Это лучше посмотреть на примере:

```
export default class Order_ extends CoreClass {
   private initSuper: (params) => Promise<IError>
   constructor(params: IObj) {
       super(params)

       this.initSuper = super.initPrototype
   }

   /**
    * Overriding the init method.
    * Here we define extraSQL for the "from_user" field to concat fio or nickname,
    * RU:
    * Переопределение метода init.
    * Здесь мы определяем extraSQL для полей "to_user" для конкатенации fio или nickname,
    * @param params
    */
   async init(params): Promise<IAPIResponse> {

       const res: IAPIResponse = await this.initSuper(params)
       if (res.code) return res

       const fromUserColProfile = this.class_fields_profile['from_user']
       const toUserColProfile = this.class_fields_profile['to_user']

       if (fromUserColProfile) {
```

```
            fromUserColProfile.extraSQL =
                `CASE
                    WHEN \`user2\`.\`use_nickname\` = 1 THEN \`user2\`.\`nickname\`
                    ELSE CONCAT_WS(' ', \`user2\`.\`lastname\`, \`user2\`.\`firstname\`,
\`user2\`.\`midname\`)
                END`

            fromUserColProfile.dependCols = [
                'from_user_id', 'use_nickname'
            ]
        }

        if (toUserColProfile) {

            toUserColProfile.extraSQL =
                `CASE
                    WHEN \`user1\`.\`use_nickname\` = 1 THEN \`user1\`.\`nickname\`
                    ELSE CONCAT_WS(' ', \`user1\`.\`lastname\`, \`user1\`.\`firstname\`,
\`user1\`.\`midname\`)
                END`

            toUserColProfile.dependCols = [
                'to_user', 'use_nickname'
            ]
        }

        return res
    }
```

7.2.2.33.3.2. Этот механизм позволяет работать с этим полем как если бы оно хранилось в базе физически (ну или было б сформировано во VIEW), как уже в конечных компонентах (таблицах/формах/фреймах), так и при получении и работе с данными в других методов на стороне бэкенда.

7.2.2.34. default_value. Позволяет указать значение по умолчанию для поля. То есть при создании записи, если значение для поля не передано, но у него указано default_value, то будет подставлено указанное значение.

7.2.2.34.1. Вы можете использовать это поле и для виртуальных полей, например чтобы указать значение по умолчанию для статуса (который берется из справочника статусов). Вместо того, чтобы указывать какое-то число для поля status_id, вы можете указать системное имя (в справочниках типа статус, мы всегда делаем поле “sysname”), то есть для виртуального поля “status_sysname”, мы можем указать default_value=”CREATED”, тогда при создании, система сама обратится в справочник статусов (тот что указан как from_table, для поля status_sysname), получит ID этого статуса и подставит его в поле status_id.

7.2.2.35. validation. Строка. Позволяет указать имя функции, которая будет проводить валидацию. Стандартные функции

имеются в файле functions.ts в объекте validation (isDate, notNull, number, url, email). Но можно определить свою как метод класса (this.myValidateFn) и указать в validation “myValidateFn”.

7.2.2.35.1. Как ее написать, смотрите существующие примеры. Но помимо самой функции, следует написать пример и добавить его (не трогая CoreClass) в this.validationFormats.

```
number: {
   format: '<число>',
   example: '10'
},
```

7.2.2.35.1.1. В самом классе написать this.validationFormats.myValidateFn = {format:’AABBCC’, example:’AABBCC’}.

7.2.2.35.1.2. Это требуется для того чтобы сообщение об ошибки валидации было наглядным.

7.2.2.35.2. Когда запись не проходит валидацию (по одному или нескольким полям), то метод (add/modify) вернет ошибку с объектом {message: 'Одно или несколько полей имеет неверный формат', fields: not_valid}, где not_valid - массив объектов вида {field: field, format: this.validationFormats[valFunc] || ''}.

7.2.2.36. get_formating, set_formating. Позволяет указать особую функцию для форматирования при получении и сохранении. Важно отметить, что такие функции применяются системой автоматически в зависимости от типа поля (например, дата переформатируется из формата MySQL в user friendly и обратно), но можно заменить стандартную обработку. Используется редко.

7.2.2.37. is_unique. Система следит, чтобы сочетание полей отмеченных этой галочкой было уникальным. Также можно включить дополнительное отслеживание на уровне СУБД - см. mysql_unique.

7.2.2.38. hint. Здесь можно указать текст подсказки, который будет показан при наведении на поле (колонку) в клиентских компонентах (таблицах/формах/фреймах). Может использоваться и в кастомных компонентах.

7.2.2.39. min_value, max_value, value_step. Определяет пределы и шаг для числовых полей, однако в текущей версии нигде не используется (почему то ни в таблицах, ни в

формах/фреймах, ни на бэкэнде). При необходимости будет доработано.

7.2.2.40. min_datetime_value, max_datetime_value, default_datetime_value. Позволяет определить минимальные и максимальные границы для полей формата даты и даты/время, а также указать дефолтное значение. Реализовано в клиентском компоненте фреймов и позволяет использовать подстановки now (текущая дата и время) или now_date (текущая дата).

7.2.2.40.1. При необходимости могут быть доработаны прочие клиентские компоненты, а также реализована проверка на бэкенде.

7.2.2.41. calender_options. Позволяет указать кастомные настройки для календаря (компонент выбора даты и время). Документация здесь: https://flatpickr.js.org/options/ (версия может быть устаревшей).

7.2.2.42. is_inherit. Позволяет указать, что это поле наследуемое. Это значит, что если нет своего значения, то в это поле может быть возвращено значение этого же поля ближайшего родителя, у которого есть это значение. Это будет работать если для класса/клиентского объекта включен режим см. enableInheritValues.

7.2.2.43. is_class_field, is_co_field. Системные поля.

7.2.2.44. save_log, always_save_log. Отмечают поля, для которых необходимо вести историю изменений. См. Механизм истории.

7.2.2.44.1. Поля дополнительно отмеченные галочкой always_save_log, будут сохраняться не только когда они меняются, а при любом любых других отслеживаемых полей. Не факт, что это нужно в текущей реализации (раньше таким признаком отмечалось поле даты, сейчас дата и так логируется).

7.2.2.44.2. Чтобы механизм работал, нужно создать дополнительную таблицу для класса. См. описание механизма.

7.2.2.45. depend_column, is_depend_where_func. Используется для клиентского компонента фреймов. Позволяет указать, от какого поля зависит это поле (оно будет недоступно, пока не

будет указано исходное). Также можно указать имя функции, которая позволит сформировать where, которое будет наложено на запрос для получения выпадающего списка этого поля (опираясь на данные, установленные в зависимом поле). Пример таких функций есть в frame_example.

7.2.2.46. frame_for_edit_values. Системное поле, позволят (для динамических полей) указать фрейм который будет открыт для редактирования значения.

7.2.2.46.1. Реализована поддержка в таблицах.

7.2.2.46.2. Возможно, можно будет использовать не только для динамических полей, но и для других, если хочется сделать кастомный редактор поля.

8. Создание/редактирование класса (сущности - таблица + профайлы + js класс + ts структуры).

8.1. Работа с tables.json

8.1.1. Создание новой сущности начинается с models/system/**tables.json** или отдельных json файлов в models/system/tables. Частично это уже описано в статье.

8.1.1.1. Практика показывает, что с декомпозированный tables.json не всегда удобно работать, поэтому выносить описание классов в отдельный файл имеет смысл, если это несколько классов имеющих общее назначение. Точно не стоит создавать отдельный файл под каждый отдельный класс, это очень не удобно.

8.1.1.2. Реализованные в ядре классы описаны в tablesCore.json или вынесены в отдельные файлы в models/system/tables (например, order.json содержит все что связано с заказами).

8.1.2. Сам tables.json содержит **начальное (неполное)**, описание класса и его полей. Остальные настройки самого класса проставляются автоматически и доступны для редактирования через интерфейс системы. Таким образом, вам необходимо задать самые базовые настройки и короткое описать только тех полей, которые вам нужны. Помимо описанных вами полей система автоматически добавит несколько системных полей, которые однако могут быть использованы в пользовательской логике, например поле created.

8.1.3. Автоматически добавляемые поля.

8.1.3.1. Здесь мы немного повторяемся со статьей “Автоматически создаваемые поля”.

8.1.3.2. created. Тип datetime. Автоматически проставляется при создании любой записи.

8.1.3.2.1. created_by_user_id. Автоматически проставляется ID пользователя, создавшего запись.

8.1.3.2.2. created_by_user. Виртуальное поле, по умолчанию не видимое, а значит и не участвующее в запросах. Поле подтягивает lastname из таблицы пользователей. При необходимости вы можете его сделать видимым и пользоваться.

8.1.3.3. updated. Тип datetime. Автоматически проставляется при изменении любой записи.

8.1.3.3.1. last_edited_by_user_id. Автоматически проставляется ID пользователя, изменяющего запись.

8.1.3.3.2. last_edited_by_user. Виртуальное поле, по умолчанию не видимое, а значит и не участвующее в запросах. Поле подтягивает lastname из таблицы пользователей. При необходимости вы можете его сделать видимым и пользоваться.

8.1.3.4. deleted. Тип datetime. Автоматически проставляется при удалении записи.

8.1.3.4.1. deleted_by_user_id. Автоматически проставляется ID пользователя, удаляющего запись.

8.1.3.4.2. deleted_by_user. Виртуальное поле, по умолчанию не видимое, а значит и не участвующее в запросах. Поле подтягивает lastname из таблицы пользователей. При необходимости вы можете его сделать видимым и пользоваться.

8.1.3.4.3. Иерархические поля.

8.1.3.4.3.1. Система может определить, что таблица ссылается сама на себя и следовательно является иерархической. Для таких таблиц будут автоматически созданы следующие поля:node_deep, parents_count, children_count. См. описания базовых концепций ядра.

8.1.4. Заполнение tables.json

8.1.4.1. Заполняйте, только самое базовое, что необходимо. Подробнее см. ниже.

8.1.4.2. Каждый класс, это json объект со следующей структурой:

8.1.4.2.1. Ключ объекта в файле, это имя класса.

8.1.4.2.1.1. Оно должно удовлетворять требованиям наименований таблиц в СУБД (в данной версии это MariaDB). Так как класс в ядре это и таблица в базе и классв javascript (см. базовые концепции), то именно по этому имени разработчик сможет вызывать базовые методы (и переопределять их или писать свои если потребуется). При этом в js (а точнее в ts), класс будет конечно называться с Большой буквы, согласно требованиям JS.

8.1.4.2.1.2. Этот ключ должен полностью совпадать с полем name в объекте, в поле “profile” (example.profile.name должно быть “example”, а user.profile.name должно быть “user”).

8.1.4.2.2. Поле profile.

8.1.4.2.2.1. Это объект, содержащий настройки самого класса (а не его полей). Здесь в принципе могут быть любые поля из описания в статье “Поля профайла самого класса”, но обычно описаны только 3-5 полей.

8.1.4.2.2.2. name. Уже упомянутый выше (должен совпадать с ключом всего JSON объекта класса).

8.1.4.2.2.3. name_ru. Не обязан быть на русском языке, не смотря на наименование поля. Это user friendly наименование класса. Именно оно будет отображено в клиентских компонентах таблиц и форм. Именно оно будет подставляться в генерируемые сообщения при манипуляциях с данными этого класса (см. ending).

8.1.4.2.2.4. ending. Используется в русском языке для добавления склонения при формировании сообщений при манипуляциях с данными.

8.1.4.2.2.4.1. Например, для сущности женского рода, “Локация”, нельзя написать “Локация изменен” (при изменении записи в таблице или форме), а нужно написать “Локация изменена”. Для этого мы пишем букву “а” в поле ending и система начинает формировать сообщения правильно.

8.1.4.2.2.5. server_parent_table и server_parent_key. См. описание в основной статье.

8.1.4.2.3. Поле structure.

8.1.4.2.3.1. Это объект объектов, где каждый объект это поле (колонка) класса.

8.1.4.2.3.2. type и length

8.1.4.2.3.2.1. Требует обязательного описания типа поля и его длины (для некоторых типов), в формате СУБД (MariaDB).

8.1.4.2.3.2.2. Наиболее распространенные:

- **type:”bigint”, length”20”**. Часто используется для поля “id”. Вы можете много где в существующем коде встретить, что он используется, в том числе, и для таблиц типа справочников, это связано с тем, что мы не уделяли этому особого внимания. Лучше используйте те типы полей, которые соответствуют ожидаемому объему данных.
    - Обратите внимание, что если вы указали id в справочнике статусов как tinyint, а в другой таблице у вас есть поле связи status_id (ссылающиеся на таблицу статус), то status_id должен иметь тот же тип.
    - Для удобства предоставляю здесь таблицу диапазонов числовых типов MariaDB (по версии Copilot):

Тип данных	Размер (байт)	Диапазон SIGNED	Диапазон UNSIGNED
TINYINT	1	-128 до 127	0 до 255
SMALLINT	2	-32,768 до 32,767	0 до 65,535
MEDIUMINT	3	-8,388,608 до 8,388,607	0 до 16,777,215
INT / INTEGER	4	-2,147,483,648 до 2,147,483,647	0 до 4,294,967,295
BIGINT	8	-9,223,372,036,854,775,808 до 9,223,372,036,854,775,807	0 до 18,446,744,073,709,551,615

    - Не используйте tinyint в сочетании с length=1 как числовое поле, так как оно используется как булевое (чекбокс).

    - Length для числовых полей определяет кол-во видимых символов и влияет на форматирование вывода в некоторых случаях (например, при использовании ZEROFILL).
- **type:"tinyint", length"1"**. Используется как булевое значение (чекбокс). Распознается системой именно как булевое, а не числовое. Из этого следует что система делает приведение типов, а также ставит по умолчанию правильный тип редактора.
- **type:"varchar", length"255"**. Используется для коротких текстов. Length может быть указан и другой, он указывает сколько максимально может храниться символов в поле. Для кодировки utf8mb4 (до 4 байт на символ) максимум — VARCHAR(16383).
    - Подбирайте length в соответствии с данными которые собираетесь хранить, так как у MariaDB есть ограничение на максимальный размер одной строки (65535 байт (для utf8mb4 один символ может занимать до 4 байт)), то есть если у вас много полей varchar вы можете достигнуть этого ограничения. Для полей с большим объемом, лучше используйте TEXT, но по нему не строится индекс и работает медленнее.
- **type:"text"**. length указывать не нужно. Если нужны другие размерности, то вы можете использовать их, например longtext.
- **type:"`DECIMAL(15,2)`"**. Могут быть и другие цифры. length указывать не нужно. Используется для дробных чисел, например для суммы денег/баллов или процентов (DECIMAL(5,2)).
- **type:"datetime"**. length указывать не нужно. Используется для хранения даты и время. В базе дата и время хранятся в стандартном для MariaDB формате (YYYY-DD-MM HH:mm:ss), однако ядро автоматически занимается форматированием, так что метод get будет возвращать в user friendly формате (DD.MM.YYYY HH:mm:ss). Передавать значения в методы (add/modify и пользовательские) нужно в том же user friendly формате (DD.MM.YYYY HH:mm:ss).
- **type:"date" и type:"time"**. По аналогии с **datetime**, только соответствующий формат.

8.1.4.2.3.2.3. Пишите маленькими буквами, кроме DECIMAL.

8.1.4.2.3.2.4. Для поля "id" обычно также указаны следующие параметры: "notNull": true, "autoInc": true, "primary_key": true, но в текущей версии ядра их можно не указывать, они будут подставлены автоматически. **Поле "id" вообще можно не указывать - будет создано автоматически**.

8.1.4.2.3.3. name

8.1.4.2.3.3.1. Это user friendly наименование поля. Именно так будет подписано поле в клиентских компонентах формах, фреймах и таблицах.

8.1.4.2.3.3.2. Если не указать, то в качестве name будет имя самого поля.

8.1.4.2.3.4. is_virtual, from_table, keyword, return_column, join_table, concat.

8.1.4.2.3.4.1. См. описание полей профайла.

8.1.4.2.3.5. hint

8.1.4.2.3.5.1. Также можно часто встретить. Позволяет определить подсказку для поля. См. описание полей профайла.

8.1.4.2.3.6. Прочие поля

8.1.4.2.3.6.1. Можно и другие поля профайла описать в tables.json, но большая часть настроек делается уже в интерфейсе или оставляется по умолчанию.

8.1.4.2.3.6.2. См. описание полей профайла.

8.1.4.3. Синхронизация из tables.json

8.1.4.3.1. Когда нужный вам класс прописан в tables.json (или любом другом подходящем для этого файле (models/system/tables/*.json), его необходимо синхронизировать.

8.1.4.3.2. В процессе синхронизации ядро создает или дополняет (если уже был создан) таблицу в базе данных, заполняет или дополняет профайл самого класса и профайлы его полей.

8.1.4.3.3. Настройки класса указанные в tables.json перезапишут существующие в профайле, но только те,которые явно прописаны.

8.1.4.3.4. В процессе синхронизации полный профайл класса, полей класса, а также его клиентских объектов и их полей, будет сохранен в соответствующий одноименный (с классом) json файл в DB/migration.

8.1.4.3.4.1. Для некоторых системных классов (например “menu”), а также всех справочников вида “ds_<имя_справочника>”, также будет сохранены данные (непосредственно записи из этих таблиц).

8.1.4.3.4.2. На основе этих файлов другой разработчик сможет получить последние обновления структуры базы и настроек профайлов. См. “Механизм миграции базы между разработчиками”.

8.1.4.3.5. Чтобы запустить синхронизацию, необходимо зайти в интерфейс, в меню System -> Classes.

8.1.4.3.5.1. Далее, если класс еще не создан, создать запись в таблице (кнопка “Создать” над таблицей), заполнить только одно поле “name”. Значение должно соответствовать ключу создаваемого вами класса в tables.json.

8.1.4.3.5.2. После создания записи или если она уже была создана ранее, вызываем контекстное меню правой клавишей по этой строке и выбираем пункт “Синхронизировать с tables.json”

8.2. Создание файла класса, работа с кодом

8.2.1. Прежде всего, как уже говорилось выше, после создания класса в tables.json и синхронизации его в Class_profile, класс становится полностью работоспособен. То есть у него уже есть все базовые методы, таблица в БД, свой пункт меню, который выведет в интерфейс таблицу (для ds_ создается в Dictionary_system, для d_ в Dictionary или Basic_data, для остальных в Temporary), а также его методы можно вызывать через внутреннее (и внешнее) api, как из методов на бэкенде, так и с клиента.

8.2.1.1. Также при синхронизации создается файл с описанием структур typescript, связанных с этим классом и в нем автоматически создается/обновляется интерфейс для полей этого класса, например для класса Crm_user: ICrm_userDataRow, который может пригодится для методов работающих с записями этого класса. Он располагается в classes/_structures/<class_name>.ts

8.2.2. Для работоспособности файл класса вам создавать не нужно, если вы не хотите дописать новые методы или переопределить

базовые. Однако в текущей версии, вызов методов какого либо класса из других методов, других (или того же) классов, осуществляется через конструкцию вида:

```
res = await r.api(Deal, 'getById', prepareP(p))
```

где “Deal” это сам класс импортируемый из файла класса. Это необходимо, чтобы typescript мог проверять параметры.

Таким образом, **нам необходимо создать файл класса**.

8.2.3. Чтобы создать файл класса, зайдите в интерфейсе системы в меню System->Classes, найдите нужный класс и в контекстном меню по этой строчке выберите “Создать файл класса.”

8.2.3.1. Файл создастся в classes/<Class_name>.ts. Его вручную необходимо добавить в git.

8.2.3.2. Созданный класс вы можете импортировать для использования в синтаксисе внутреннего API (**res = await r.api(<Class_name>, '<methodName>', prepareP(p))**). Но **не создавать** его инстанс вручную для вызова его методов, так как для этого используется именно внутреннее api. см. базовые концепции.

8.2.3.3. В нем же вы можете писать свои методы, или переопределять базовые.

9. Создание Клиентских Объектов.

9.1. Прежде всего, давайте определимся, для чего нам создавать клиентский объект. Чаще всего это связано с необходимостью сделать особые настройки для различных клиентских интерфейсов. Отсюда и название - клиентские объекты. Когда вы хотите одну и ту же таблицу вывести и в основном меню и, например, внутри формы со срезом данных ограниченных этой родительской формой (например, таблица Позиций Заказа в форме Заказа), или две разных формы, более полную и сокращенную, с разным набором полей - вам нужны клиентские объекты.

9.2. Создание профайла и синхронизации

9.2.1. Чтобы создать клиентский объект, нужно зайти в меню System -> Client objects, в таблице создать новую запись, указать имя, выбрать класс, для которого делаете КО. После создания записи, через контекстное меню выбрать пункт Синхронизация с классом.

9.2.2. Рекомендации по наименованию

9.2.2.1. В первую очередь цель правил, сделать так, чтобы из наименованию было примерно понятно для чего используется этот объект.

9.2.2.2. Наименование должно быть уникально.

9.2.2.3. Для таблиц.

9.2.2.3.1. Должно начинаться с “table_”, для таблиц размещаемых в основном меню, то есть в главном интерфейсе, или “tbl_”, для таблиц встраиваемых в формы/фреймы/модальные окна.

9.2.2.3.2. В некоторых случаях, например, когда таблица кастомизируется и используется, например, как набор фильтров, а не непосредственно таблица, клиентский объект вполне может называться как-нибудь еще, например “dashboard_…”.

9.2.2.3.3. После префикса обычно идет имя класса, а далее может быть указано уточнение. Например “table_order_full” или “table_user_admins_only”.

9.2.2.4. Для Фреймов и Форм.

9.2.2.4.1. Аналогично таблицам, только с другими префиксами. Так как формы не могут быть встроены друг во что либо, для них нет краткой формы, а есть только полная “form_”, а для фреймов может быть “frame_” или “frm_”. Опять таки, это лишь рекомендации, и если вы считаете, что более подходящим наименованием будет другое, не соответствующее им, это допустимо.

9.2.3. Синхронизация с классом

9.2.3.1. При синхронизации будут добавлены недостающие поля. Кроме этого, в диалоговом окне будут предложены дополнительные опции, а именно синхронизировать все или перечисленные настройки (профайл полей клиентского объекта) и удалить несуществующее (удаленные из класса поля).

9.2.3.1.1. Галочка “Синхранизировать поля (профайл полей, например filter_type_id)”. Позволяет всем полям клиентского объекта (id, name, sysname, description… те что есть у этого клиентского объекта) перенести

выбранные настройки (поля) из класса. Чтобы это сделать надо поставить указанную галочку и оставить в поле ввода “*”.

9.2.3.1.1.1. Можно перенести только некоторые настройки, для этого также надо поставить галочку, но в поле ввода перечислить через запятую наименования профайлов полей (на английском, как они указаны в tablesCore.json в class_fields_profile), которые, вы хотите чтобы были перенесены. Например “editable, filter_type_id”.

9.2.3.1.2. Галочка “Удалять поля”. Позволяет удалить поля, если они были удалены в классе. Раньше были механизмы, которые подразумевали возможность у клиентского объекта таких полей, которых нет у класса, поэтому это галочка не стоит по умолчанию в некоторых проектах. Сейчас галочка стоит по умолчанию.

9.2.3.1.2.1. Удалять физически, тоже стоит по умолчанию. Оставьте ее поставленной. Она отключает soft deletion для этих записей.

9.2.4. Синхронизация с классом (обновить класс)

9.2.4.1. Позволяет наоборот, влить настройки сделанные на уровне клиентского объекта, в класс. Это бывает полезно, когда вы настроили один клиентский объект (привели его к user friendly виду - скрыли лишние поля, дали понятные наименования, прописали подсказки, настроили фильтры…), а теперь хотите, чтобы все эти настройки попали в класс, чтобы и другие клиентские объекты могли получить эти настройки при создании или синхронизации.

9.2.4.1.1. Также можно указать перенос всех настроек (“*”), или перечислить нужные через запятую в поле ввода.

9.2.5. Синхронизировать в другой клиентский объект

9.2.5.1. По аналогии с предыдущем пунктом (“Синхронизация с классом (обновить класс)”) позволяет перенести все или часть настроек, но уже не в класс, а в другой КО. Для этого в диалоговом окне есть поле, в которое нужно ввести ID КО, в которое надо влить настройки.

9.2.6. Установить значение профайла массово

9.2.6.1. Иногда, в разрабатываемой системе имеется множество схожих клиентских объектов на одном классе (например для разных ролей), и обновлять их всех по одному, если нам нужно внести изменения для всех, довольно утомительно. Тогда можно сделать это массово. Для этого, необходимо выбрать необходимые строки в таблице (те клиентские объекты которые хотите обновить), выбрать в контекстном меню этот пункт, далее, в диалоговом окне, указать 3 поля:

- Какое поле меняем (это column_name, так как он записан в tables.json).
- Какую настройку мы хотим установить (это может быть галочка visible или тип редактора (type_of_editor_id) или любая другая настройка).
- В какое значение привести эту настройку.

9.2.6.2. Синхронизация сработает для всех выбранных клиентских объектов.

9.2.6.3. Также там есть кнопки Load, Save, Clear, которые позволяют сохранить и восстановить введенные данные, чтобы каждый раз не вводить все заново.

9.3. Создание компонентов интерфейса

9.3.1. Здесь речь пойдет о компонентах встроенного в ядро фронтенда и его базовых компонентах: Таблицах, Формах, Фреймах.

9.3.1.1. Внимание! Чтобы вывести клиентский компонент (таблицу/форму/фрейм), нет необходимости создавать отдельный Клиентский объект (System->Client objects). Если Клиентский объект не указан, то будет использоваться профайл класса.

9.3.1.2. На этих компонентах, плюс некоторые вспомогательные, построен весь интерфейс встроенного фронтенда. Выше это уже упоминалось, однако повторюсь, что данный фронтенд, в зависимости от проекта может используется как только бэкофис, бэкофис и основной пользовательский интерфейс(для ERP систем он хорошо подходит), или вообще не используется, если это чисто бэкенд сервис.

9.3.1.3. Все эти компоненты плотно интегрированы с бэкэндом, а именно они загружают и используют профайл Класса/КлиентскогоОбъекта и его полей, на их основе собирают интерфейсы, дополнительные компоненты, такие как редакторы, собирают параметры для различных

запросов к бэку, передают бэкенду информацию о взаиморасположение относительно друг друга, что позволяет ему (бэкенду) автоматически добавлять различные срезы данных.

9.3.2. Таблицы.

9.3.2.1. Таблицы прежде всего предназначены для отображения и манипуляций с данными, но также являются точкой входа/старта для различных кастомных функций реализующих бизнес-логику проекта. Это может быть открытие формы, где уже будет стандартный или кастомный интерфейс, реализующий необходимую функциональность, или открывающий модальное окно с интерфейсом или просто диалог с подтверждением действия и отправкой команды на сервер. Также могут быть реализованы кастомные кнопки над таблицей или в скрытом по умолчанию дополнительном меню (там же где Выгрузка в excel/json, открывается через звездочку). Вы можете кастомизировать таблицу практически как угодно, например скрыв тело таблицы и оставив только функционал фильтров, который вы можете использовать, например, для отправки своего запроса на сбор аналитических данных (с этими фильтрами) и дальнейшей отрисовки графиков по этим данным.

9.3.2.2. Как создать

9.3.2.2.1. Создание профайла

9.3.2.2.1.1. Как уже говорилось выше, вы можете не создавать отдельный профайл для таблицы, а использовать профайл класса.

9.3.2.2.1.2. Если же вам нужны особые настройки профайла, отличные от настроек профайла класса, то создайте Клиентский объект, как это описано в разделе “Создание Клиентских Объектов”.

9.3.2.2.2. Размещение в интерфейсе

9.3.2.2.2.1. В основном меню

9.3.2.2.2.1.1. Необходимо создать пункт меню типа “Элемент” в редакторе меню (System->Menu editor), указать Родительский элемент (в каком пункте разместить), выбрать Класс и при

желании Клиентский объект (если выбран клиентский объект, он должен быть основан на том же классе, что и выбранный класс).

9.3.2.2.2.1.2. Такой элемент появится в меню и при его выборе будет открыта указанная таблица.

9.3.2.2.2.1.3. В коде ничего писать не потребуется.

9.3.2.2.2.2. Внутри формы или фрейма (в верстке).

9.3.2.2.2.2.1. В верстке формы или фрейма вы можете указать специальный div элемент с определенным классом и дата атрибутами, тогда при загрузки формы/фрейма в этом div будет загружена соответствующая таблица.

```
<div data-tbls="CLASS_NAME.TABLE_CO_NAME" class="fn-child-tbl-holder marTop10"></div>
```

9.3.2.2.2.2.1.1. class: **fn-child-tbl-holder**

9.3.2.2.2.2.1.2. data-tbls: **"CLASS_NAME.TABLE_CO_NAME"**, где CLASS_NAME это имя класса, а TABLE_CO_NAME - имя клиентского объекта. Имя клиентского объекта может быть не указано, тогда атрибут будет выглядеть так "CLASS_NAME."

9.3.2.2.2.3. Кодом, в любой контейнер.

9.3.2.2.2.3.1. Чтобы создать таблицу, нужно создать новый инстанс MB.TableN и передать ему параметры, после вызвать метод create передав параметром контейнер (jQuery).

Пример:

```
const req_holder = dashboard.parentBlock.find('.dbrd-requests-container');

const req_table = new MB.TableN({
   name: 'Заявки',
   client_object: 'table_request_work_dashboard',
   class: 'request_work',
   id: MB.Core.guid(),
   parentObject: frameInstance,
   parent: frameInstance,
   parent_id: frameInstance.data.data.id,
   destroy_on_reload: true,
```

```
    externalWhere: []
});


req_table.create(req_holder, function () {
    // console.log('dashboard table rendered');
});
```

9.3.2.2.2.3.1.1. name. Ни на что не влияет, позволяет лучше ориентироваться в коде и инстансах в MB.Tables.tables.

9.3.2.2.2.3.1.2. class и client_object. Соответственно указывает на Класс и Клиентский объект (опционально)

9.3.2.2.2.3.1.3. id. Должен быть уникален.

9.3.2.2.2.3.1.4. parentObject, parent, parent_id, destroy_on_reload. Если таблица встраивается в форму или фрейм, тогда имеет смысл указать ее как родителя и указать id открытой в этой форме или фрейме записи. Это позволит таблице (при указании в профайле parent_key) накладывать ограничения на загружаемые данные, фильтруя по этому ключу. Также на них основываются другие внутренние механизмы.

9.3.2.2.2.3.1.4.1. Параметр destroy_on_reload определяет уничтожать ли таблицу при перезагрузке родительского компонента.

9.3.2.2.2.3.1.5. Другие параметры, необязательные, например externalWhere, позволяет передать массив условий в стандартной для клиента форме, но его можно наложить

и в профайле клиентского объекта.

9.3.2.2.3. js файл расширения функционала таблицы

9.3.2.2.3.1. При загрузки таблицы может быть также загружен и исполнен js файл, в котором можно получить инстанс таблицы и расширить ее функционал. Чтобы подключить такой файл, его требуется создать в директории public_src/html/tables/require и назвать также, как называется Клиентский объект или, если он не указан, “table_<имя_класса>”, а также в профайле Клиентского объекта (или класса) установить галочку “Additional functionality”.

9.3.2.2.3.2. За основу такого файла можно взять public_src/html/tables/require/_gocore/table_example.js

9.3.2.2.3.3. Как писать контекстное меню

9.3.2.2.3.3.1. В файле table_example.js уже имеется заготовка под контекстное меню.

9.3.2.2.3.3.2. В этом файле у вас имеется получение инстанса текущей таблицы и ему может быть добавлен массив tableInstance.ct_instance.ctxMenuData, где каждый элемент это объект описывающий пункт контекстного меню.

9.3.2.2.3.3.3. Поля пункта контекстного меню

9.3.2.2.3.3.3.1. name. Должно быть уникально в данном массиве.

9.3.2.2.3.3.3.2. title. Наименование отображаемое в интерфейсе.

9.3.2.2.3.3.3.3. disable. Функция, возвращающая булевое значение. Если возвращено true, то пункт меню будет недоступен. В этой функции можно, например на данные выбранной строки решить,

применимо ли для нее это действие. Получить текущую строку можно через tableInstance.ct_instance.selectedRowIndex. См. поле callback.

9.3.2.2.3.3.3.4. callback. Функция, которая будет исполнена при клике на выбранный пункт (если он активен). Вы можете получить выбранную строку, а также все выбранные строки (и вообще все данные, настройки таблицы, выбранные фильтры и прочее).

```
{
   name: 'option2',
   title: 'Другое',
   disabled: function(){
       return false;
   },
   callback: function(){
       const row = tableInstance.ct_instance.selectedRowIndex
       const dataRow = tableInstance.data.data[row]
       const id = dataRow.id

       // Все выбранные
       const selected = tableInstance.ct_instance.selection2.data
   }
}
```

9.3.2.2.3.4. Как писать другой функционал

9.3.2.2.3.4.1. Как показано в примере вы можете получить доступ к инстансу таблицы, а следовательно ко всем его настройкам, данным и текущему состоянию, например фильтрам или тексту в поле быстрого поиска или текущей странице и прочее. Подробнее пока не описано, но вы можете изучить содержимое объекта в рантайме или загрузив таблицу, в консоле браузера набрать MB.Tables.tables (массив загруженных таблиц).

9.3.2.2.3.4.2. Далее вы можете писать любую логику в соответствии с вашей задумкой, пользуясь существующими в ядре компонентами или подключая внешние (распространенные или

самописные) или вообще загрузить туда отдельное приложение на react или другом фреймворке передав туда нужные вам параметры.

9.3.3. Фреймы.

9.3.3.1. Фреймы предназначены для отображения любого контента, позволяя встраивать их в любые другие компоненты интерфейса, в том числе и в самих себя. Могут быть пустые фреймы, однако обычно они привязаны к Клиентскому объекту, то есть загружают профайл Класса/Клиентского объекта и его полей, а также загружают данные указанной записи.

9.3.3.2. Разработчик может реализовать любую верстку, но при этом использовать шаблонизацию подставляя загруженные поля, с редактором или только значение. При подстановке значений с редактором, отображение будет выполнено с учетом настроек редактора для этого поля в профайле, а также с учетом настройки редактируемости. Для не редактируемого поля не будет подключен соответствующий редактор, однако все остальные атрибуты поля будет (например, подсказка).

9.3.3.2.1. Базово, обычно используется верстка bootstrap (вроде еще v3). Но можете писать как хотите.

9.3.3.3. Помимо вывода полей Класса/КО, могут быть также встроены таблицы или другие фреймы, об этом упоминалось в описании таблиц.

9.3.3.3.1. Чтобы встроить таблицу, необходимо вывести div элемент с определенным классом и дата атрибутами, тогда при загрузки формы/фрейма в этом div будет загружена соответствующая таблица. См. описание “Таблицы. Внутри формы или фрейма (в верстке)".

```
<div data-tbls="CLASS_NAME.TABLE_CO_NAME" class="fn-child-tbl-holder marTop10"></div>
```

9.3.3.3.2. Чтобы встроить фрейм, также необходимо вывести div с определенным классом и дата атрибутом.

```
<div data-frame="CLASS_NAME.FRAME_CO_NAME" data-ids="{+{id}+}" data-new_if_not_id="true"
class="fn-child-frm-holder"></div>
```

9.3.3.3.2.1. data-frame. Указать имя класса и КО через точку. КО можно не указывать.

9.3.3.3.2.2. class. Должен содержать класс “fn-child-frm-holder”

9.3.3.3.2.3. data-ids. Если указано “{+{id}+}” то будет подставлено ID записи, той формы или фрейма в которое вставляется.

9.3.3.3.2.4. data-new_if_not_id. Указывает, что если id не передан, то запись открывается для создание (как новая строка в таблице). Значение будет истинно, только если строка равна “true”, во всех остальных будет false.

9.3.3.4. Как создать

9.3.3.4.1. Создание профайла

9.3.3.4.1.1. Как уже говорилось выше, вы можете не создавать отдельный профайл для таблицы, а использовать профайл класса.

9.3.3.4.1.2. Если же вам нужны особые настройки профайла, отличные от настроек профайла класса, то создайте Клиентский объект, как это описано в разделе “Создание Клиентских Объектов”.

9.3.3.4.2. Верстка и js

9.3.3.4.2.1. Файлы фрейма необходимо создать в кодовой базе и разместить в public_src/html/frames. Имя директории это и есть имя фрейма, а внутри должны быть одноименные html и js файлы. Опционально может быть размещен ccs файл. Как правило это имя совпадает с именем клиентского объекта или имеет вид “frame_<class_name>”.

9.3.3.4.2.2. Вы можете писать любую верстку, как уже говорилось выше.

9.3.3.4.2.3. В js файле вы также можете как угодно организовывать код, но есть несколько полезных вещей, которые следует знать.

9.3.3.4.2.3.1. Весь код оборачивается в самовызываемую функцию (function(){})().

9.3.3.4.2.3.2. Чтобы получить инстанс фрейма, напишите в самом начале (уже имеется в примере):

```
var frameID = MB.Frames.justLoadedId;
var frameInstance = MB.Frames.getFrame('frame_example', frameID);
```

9.3.3.4.2.3.2.1. Важно! Строковый первый параметр в методе getFrame нужно заменить на имя вашего фрейма.

9.3.3.4.2.3.3. Далее по аналогии с таблицами вы можете использовать этот инстанс. Вы можете получить настройки профайлов Класса/КО и его полей, загруженные данные, состояния (например измененные поля), родителя при наличии. А также определить функции, которые вызываются после определенных событий фрейма, таких как afterLoad, afterReload, afterAdd.

9.3.3.4.2.3.4. Часто для объединения всех задач конкретного фрейма используется объект frameEditor у которого есть поля для хранения состояния и различные методы, например load или setHandlers. Это лишь пример, и код можно организовыввать по другому (более современно) если в этом есть какая-либо необходимость.

9.3.3.4.2.3.5. Смотрите пример, и используйте как референс для копипаста C:\NET\CCS.ARLAN_SALON\public_src\html\frames\frame_example

9.3.3.4.3. Размещение в интерфейсе

9.3.3.4.3.1. В основном меню

9.3.3.4.3.1.1. Необходимо создать пункт меню типа “Фрейм” в редакторе меню (System->Menu editor), указать Родительский элемент (в каком пункте разместить), выбрать Класс и при желании Клиентский объект (если выбран клиентский объект, он должен

быть основан на том же классе, что и выбранный класс).

9.3.3.4.3.1.2. Такой элемент появится в меню и при его выборе будет открыт указанный фрейм в основной контентной области.

9.3.3.4.3.1.3. В коде ничего писать не потребуется.

9.3.3.4.3.2. Внутри формы или фрейма (в верстке).

9.3.3.4.3.2.1. В верстке формы или фрейма вы можете указать специальный div элемент как описано в описании к фреймам, см выше.

9.3.3.4.3.3. Кодом, в любой контейнер.

9.3.3.4.3.3.1. Чтобы создать фрейм кодом, нужно вызвать MB.Frames.createFrame, передав параметры и callback функцию. Если это делается внутри какого нибудь другого фрейма или формы, то при ее обновлении закрытии, надо уничтожать и дочерний созданный фрейм.

9.3.3.4.3.3.2. Также фрейм может создаваться при, например, переключении выбранного пункта в списке или выбранной ноды в дереве. Ниже, рассмотрим пример, который имеется в frame_example.js. Он создает соответствующий фрейм при клике на ноду, предварительно уничтожив старый (при наличии).

```
holder.on('select_node.jstree', function (e,a) {
   var frame_class = a.node.original.item._content_class;
   var frame_co = a.node.original.item._content_co;

   const obj = {
       container:frameEditor.frame_content,
       class:frame_class,
       client_object:frame_co,
       parent:frameInstance,
       ids:[a.node.original.item._id],
       name:frame_co
   }
   if (frameEditor.current_frame && typeof frameEditor.current_frame.remove
==='function') {
       frameEditor.current_frame.remove({remove_from_parent: true});
   }

   MB.Frames.createFrame(obj, (err, frame)=>{
       frameEditor.current_frame = frame;
```

```
        if (err) return console.error('Не удалось создать фрейм',err);

    });
});
```

9.3.3.4.3.3.2.1. Здесь мы берем нужный класс и клиентский объект из пункта меню/ноды, по которой кликнули, если имеется старый фрейм в frameEditor.current_frame уничтожаем его, потом создаем новый и сохраняем в frameEditor.current_frame.

9.3.3.4.3.3.2.2. Поля по аналогии с таблицей. Вызывает вопросы поле ids, но там просто берется нулевой элемент. Имя в принципе может быть любое, parent может быть не указан, если, например, вы создаете фрейм в модальном окне или просто в основной контентной области.

9.3.4. Формы.

9.3.4.1. Эта сущность аналогична Фреймам, только имеет собственную оболочку, позволяющую открывать одновременно несколько форм, сворачивать их в док, масштабировать и перемещать.

9.3.4.2. На самом деле именно из Форм выросли Фреймы, а не наоборот. Фреймы получили больше специфических возможностей и сейчас зачастую сценарий использования форм такой, что в их верстке просто размещается фрейм, который уже содержит всю логику. Хотя это вовсе не обязательно и если ничего особого не нужно, то большая часть функционал работает и в формах, также как и во фреймах.

9.3.5. Другие часто используемые компоненты

9.3.5.1. Вкладки

9.3.5.1.1. Вы можете найти пример вкладок в формы и фреймы, в соответствующих файлах примеров

форм/фреймов (public_src/html/forms/_gocore/form_example и public_src/html/frames/frame_example). Пока без описания.

9.3.5.2. Дерево

9.3.5.2.1. Вы можете найти пример файле примера фрейма (public_src/html/frames/frame_example). Пока без описания.

9.3.5.3. bootbox.dialog

9.3.5.3.1. Документацию можно найти в интернете по этой библиотеке.

9.3.5.3.2. Вот пример (код старый, но рабочий)

```
var dialog = bootbox.dialog({
   title: 'Сменить пароль',
   message: html,
   buttons: {
       success: {
           label: 'Установить новый пароль',
           closeButton:false,
           callback: function(){
               var old_psw = $('#old_psw').val();
               var new_psw1 = $('#new_psw1').val();
               var new_psw2 = $('#new_psw2').val();
               if (!old_psw || !new_psw1){
                   toastr.error('Заполните все поля.');
                   return false
               }
               if (new_psw1 !== new_psw2){
                   toastr.error('Пароли не совпадают.');
                   return false

               }
               var o = {
                   command: 'changePassword',
                   object: 'User',
                   params:{
                       password:old_psw,
                       new_password:new_psw1
                   }
               };

               socketQuery(o, function(res){
                   console.log(res);
               });
           }
       },
       cancel: {
           label: 'Отмена',
           callback: function(){

           }
       }
   }
});
```

9.3.5.4. select2

9.3.5.4.1. Вы можете повесить выпадающий список на любое поле, используя его профайл. Это может быть использовано, например, в модальном окне, где пользователя просят выбрать, например, организацию. Указав нужные параметры, выпадающий список будет обращаться в базу со всеми соответствующими ограничениями по доступу к данным, фильтрам и настройкам.

9.3.5.4.2.

9.3.5.5.

10. Логирование

10.1. Введение

10.1.1. В данном разделе рассмотрим где и какие логи можно смотреть, как включать и отключать более детальное логирование и как читать выводимые логи.

10.1.1.1. Важно. Здесь описаны самые ключевые аспекты. Однако чтобы свободно ориентироваться достаточно внимательно читать, что написано в логах. В редких случаях когда информации недостаточно вы можете поискать где они используются и при необходимости временно вывести дополнительную информацию или запустится в режиме отладки, поставив точку останова в необходимом месте. Если ничего не помогает пишите нам.

10.1.2. Прежде всего у нас имеется серверный процесс и его логи, и соответственно, клиентская часть со своими логами.

10.2. Серверные логи

10.2.1. В серверную консоль выводятся много полезной информации, от старта сервера, применение конфигураций, запуска различных процессов, информации о подключениях и запросах и ответах (ответы выводятся в краткой форме, так как это может быть большой объем данных).

10.2.2. Также разумеется выводится все, что разработчик решил вывести в своих методах и модулях (через **console**). Однако как правило таких логов мало и их следует убирать после отладки, чтобы они не засоряли лог.

10.2.2.1. В методах иногда имеет смысл оставлять логи вида console.error, в тех местах, где поведение не соответствует продуманной логике. При этом здесь речь идет не о проверке параметров, а более нештатных ситуациях. Это даст возможность при сбоях иметь информацию о том, что пошло не так. Проверка параметров, как и прочие проверки, должны обрабатываться штатным механизмом выхода из функции с соответствующим ответом (интерфейс IError).

10.2.3. Некоторыми аспектами вывода можно управлять, это будет описано ниже.

10.2.4. В различных модулях, могут быть свои дополнительные настройки, которые могут включать более подробное логирование. Оно было нужно при разработке этих модулей и сейчас не нужно, но если что-то работает не так, можно посмотреть код и обнаружить какой-нибудь флаг в начале файла, который поможет лучше понять что происходит. Скорее всего это вам не понадобится.

10.2.5. Информация в логах упорядочена и имеет определенную структуру, которую мы сейчас и опишем:

10.2.5.1. Начинается логирование со старта сервера и указания какой конфигурационный файл выбран:

Configuration file selected: config.json

10.2.5.1.1. Чтобы выбрать другой конфиг его нужно указать вторым (после имени js файла) параметром при запуске: “node bin/www.js testConfig.json”

10.2.5.2. Далее устанавливаются настройки MySQL взятые из конфигурации. Сообщения вида “***INFO: cMysql.INIT. ...***”. Они выводятся через error, так же как и последующее сообщение о запуске сервера:

ERROR: 10.07.2025 12:15:07 SERVER STARTED Server time: Thu Jul 10 2025 12:15:07 GMT+0300 (Москва, стандартное время)

10.2.5.2.1. Вывод этих сообщений через поток **error** сделан намеренно, так как позволяет легко найти точку старта/перезапуска среди всех логов (этого процесса) на сервере. К тому же, с точки зрения продакшена, перезапуск сервера действительно является значимым событием, поэтому ему следует быть в отдельном потоке, который в первую очередь смотрят.

10.2.5.2.2. Далее сообщений в потоке **error** быть не должно и если они есть, то на них следует обратить внимание.

10.2.5.3. После идет информация об адресе, на котором запущен сервер

Server running at http://127.0.0.1:7011/

10.2.5.4. Далее несколько сообщений от сервиса фоновых задач. В первую очередь идет сообщение, через сколько пойдет процесс запуска, так как они запускаются отложено, чтобы не давать лишней нагрузки в самом начале, когда все переподключаются.

BJ ==> INFO Фоновые задачи начнут запускаться через 300

…
BJ ==> INFO Следующие задачи исключены из запуска конфигурационным файлом []

BJ ==> INFO Эти задачи будут запущены […]

10.2.5.5. Когда подключается клиент (по сокету), будет сообщение следующего вида: ***io.on.connection:count: 2***. Когда отключается: ***socket on disconnect 4BOQu1pNzJJvs9lnAAAL transport close***.

10.2.5.6. Обмен запросами с клиента и между методами

10.2.5.6.1. Все запросы проходят через внутренний API и попадают в логирование.

10.2.5.6.2. Информация о запросе

10.2.5.6.2.1. Запросы с клиента (будь то веб интерфейс, обращение по API или мобильное приложение) открывают цепочку запросов, а все внутренние вызовы других методов, если исходный метод еще что-то вызывает) продолжают эту цепочку, то есть становятся дочерними для вызвавших его методов. Таким образом в логах мы видим всю цепочку, кто кого вызвал.

10.2.5.6.2.2. Рассмотрим на примерах запроса.

11.07.2025 10:36:53.611 f437c7d7-db07-4ca4-876e-b0749c6e5e73: → 48DE98AA_User.get_me (fromClient) {"getRoles":false}

11.07.2025 10:36:53.612 f437c7d7-db07-4ca4-876e-b0749c6e5e73: ← 48DE98AA_User.get_me (fromClient) ←1 UserError: noAuth -4 noAuth User.get_me { message_en: 'Please log in.' }

10.2.5.6.2.3. В начале идет дата и время.

10.2.5.6.2.3.1. Не смотря на то, что серверный журнал сам логирует время записи, собственное время в логах гораздо удобнее при поиске проблем.

10.2.5.6.2.4. Далее идет SID

10.2.5.6.2.4.1. Для авторизованного пользователя вы сможете найти его сессию в системе или в БД, если она уже завершилась. Также по нему вы сможете отфильтровать логи, если захотите посмотреть только для него.

10.2.5.6.2.5. Если запрос внутренний, то указывается столько символов “_” (нижнее подчеркивание), какой уровень вложенности имеет запрос.

10.2.5.6.2.6. Далее, стрелкой указано направление. Стрелка вперед это запрос, а стрелка назад - ответ.

10.2.5.6.2.7. Потом идет ID запроса, содержащее также уникальный номер и цепочку вызовов в формате <ИмяКласса>.<ИмяМетода>. При этом если следующий в цепочке метод относится к тому же классу что и родительский, то имя класса опускается.

EA55DED2_User.get_me - Access_to_operation.loadAccess - user_role.get

или

3629393D_menu.get_menu_tree - .get (тут имя класса перед “get” опускается, так как он относится тоже к классу “menu”).

10.2.5.6.2.8. Если запрос пришел извне (не внутренний), то далее идет “(fromClient):”. По нему также можно отфильтровать, чтобы не мешали внутренние запросы.

10.2.5.6.2.9. Далее для запроса (когда он приходит) идет содержимое параметров, а если это уже ответ, то стрелка влево, время выполнения метода и текст сообщения ответа (без тела ответа).

2025 12:45:42.477 f437c7d7-db07-4ca4-876e-b0749c6e5e73: fromClient: ← DD58E88A_User.login ←134 noToastr

10.2.5.6.2.9.1. Часто можно увидеть ответ “noToastr”, его возвращают методы, которые не предусматривают вывод сообщения пользователю.

10.2.5.6.2.10. Вот как выглядит набор запросов, начиная с входа извне, продолжая внутренними запросами, и завершая ответом:

11.07.2025 10:41:00.256 f437c7d7-db07-4ca4-876e-b0749c6e5e73: → D296E26E_menu.get_menu_tree (fromClient) {}

11.07.2025 10:41:00.272 f437c7d7-db07-4ca4-876e-b0749c6e5e73: _ → D296E26E_menu.get_menu_tree - .get {"where":[{"key":"is_visible","val1":true}],"sort":"sort_no,name","limit":1000000}

11.07.2025 10:41:00.347 f437c7d7-db07-4ca4-876e-b0749c6e5e73: _ ← D296E26E_menu.get_menu_tree - .get ←75 noToastr

11.07.2025 10:41:00.348 f437c7d7-db07-4ca4-876e-b0749c6e5e73: ← D296E26E_menu.get_menu_tree (fromClient) ←92 noToastr

10.2.5.6.2.11. Система скрывает некоторые секьюрные параметры при выводе в консоль, выводя вместо них “***”, например параметр “password” в методе User.login.

10.2.5.6.2.11.1. Такие параметры перечислены в **excludedConsoleParams** в api/apiConfig и соответственно в файлах default и project. Если хотите расширить список, дополните его в файле project.ts Вы можете использовать не полное совпадение имени параметра а “начинается с” и “заканчивается на”, используя символ “*”.

10.2.5.6.2.12. Конфигурация “log:api:”

10.2.5.6.2.12.1. Вы можете настроить какую информацию выводить. Настройки располагаются в конфигурационном файле config/config.json в секции “**log:api:**”.

10.2.5.6.2.12.2. **fromClientOnly**. Позволяет отключить логирование внутренних запросов. Не отключайте на продакшене, так как это довольно ценная информация.

10.2.5.6.2.12.3. **logStart**. Выводить ли в консоль когда запрос пришел.

10.2.5.6.2.12.4. **logFinishUserOk**. Выводить ли в консоль когда запрос выполнен и отправляется клиенту, если ответ UserOk.

10.2.5.6.2.12.5. **logFinishUserError**. Выводить ли в консоль когда запрос выполнен и отправляется клиенту, если ответ UserError.

10.2.5.6.2.12.6. **logFinishMyError**. Выводить ли в консоль когда запрос выполнен и отправляется клиенту, если ответ MyError.

10.2.5.6.2.12.7. **logFinishUnknown**. Выводить ли в консоль когда запрос выполнен и отправляется клиенту, если ответ не один из вышеперечисленных типов. Такого быть не должно, заложено на всякий случай, если разработчики добавят новый тип, соответствующий IAPIResponse.

10.2.5.6.2.12.8. **logBadCommand**. Выводить ли в консоль когда запрос обращается к несуществующему методу. Не стоит включать этот параметр просто так. Такие ошибки вполне могут случаться, например когда кто-нибудь отлаживает работу с вашим сервисом по API и допускает ошибки. Хотя на продакшене можно включить, так как там никаких отладок быть не должно и если такие команды имеются, то их стоит логировать и разбираться.

10.2.5.7. Логирование SQL

10.2.5.7.1. Система позволяет логировать какие SQL запросы формируются. Это относится только к запросам вида SELECT.

10.2.5.7.1.1. Почему-то за все время работы не возникло необходимости логировать другие виды SQL запросов (именно SQL, так как мы видим что метод, например “modify” был вызван и с какими параметрами), поэтому это сделано не было.

10.2.5.7.2. В ходе своей работы ядро может делать различные дополнительные (служебные) запросы, которые

помечены параметром doNotLog. Такие запросы по умолчанию скрываются из логирования, как в api, так и запросов SQL. Это можно изменить параметром **ignoreDoNotLog**.

10.2.5.7.3. Обычный get запрос делает два запроса, один на подсчет количества по этим условиям, а второй непосредственно запрос данных. Оба эти запроса можно логировать или нет. Подробнее ниже.

10.2.5.7.4. На самом деле, ядро делает много различной работы, включая подтягивание данных из справочников по системным именам, проверок уникальности и прочего. Кроме того, система кэширования устроена довольно хитро, поэтому не всегда сходу понятно, почему система делает те или иные подзапросы (что-то из кэша, тот то из базы). Такие запросы как правило идут с флагом doNotLog, о котором уже упоминалось, и на них не стоит обращать внимания. Здесь это описано для того, чтобы вы при отладке какого-либо своего запроса, включив все логирование, были в курсе, что разные процессы могут происходить и это нормально - они обеспечивают оптимальное взаимодействие с базой, при должном уровне удобства разработки.

10.2.5.7.5. Ядро умеет логировать долгие запросы, такие запросы логируются в поток “**error**” (по умолчанию), что позволяет легко их обнаруживать и предпринимать меры. Вы можете настроить параметры для этого механизма, см. Конфигурация “log:sql:”

10.2.5.7.6. Конфигурация “log:sql:”

10.2.5.7.6.1. Вы можете настроить какую информацию выводить. Настройки располагаются в конфигурационном файле config/config.json в секции “**log:sql:**”.

10.2.5.7.6.2. **countStart**. Выводить ли в консоль SQL запрос на ПОДСЧЕТ перед его выполнением в БД.

10.2.5.7.6.3. **countFinish**. Выводить ли в консоль SQL запрос на ПОДСЧЕТ после его выполнения в БД.

10.2.5.7.6.4. **selectStart**. Выводить ли в консоль SQL запрос на ПОЛУЧЕНИЕ ДАННЫХ перед его выполнением в БД.

10.2.5.7.6.5. **selectFinish**. Выводить ли в консоль SQL запрос на ПОЛУЧЕНИЕ ДАННЫХ после его выполнения в БД.

10.2.5.7.6.6. **selectFromCache**. Выводить ли в консоль SQL запросы на ПОДСЧЕТ и ПОЛУЧЕНИЕ ДАННЫХ при получении их из кэша.

10.2.5.7.6.6.1. Для таких запросов есть только перед получением.

10.2.5.7.6.7. **ignoreDoNotLog**. Выводить выше перечисленные запросы, согласно их настройкам, в том числе и для запросов с параметром **doNotLog**. Иногда помогает отладить сложный случай.

10.2.5.7.6.8. **debugLongSql**. Включает логирование долгих запросов, даже если все остальное логирование отключено.

10.2.5.7.6.9. **longSQLTime**. Определяет в миллисекундах, больше какого времени должен выполняться запрос, чтобы считаться “долгим” и попадать в лог.

10.2.5.7.6.10. **longSQLConsoleFn**. Можно переопределить, какой метод вызывать у объекта console, при логировании долгих запросов. По умолчанию выполняется console.error(...).

10.2.5.8. Сохранение в базу данных

10.2.5.8.1. Система автоматически сохраняет некоторые ошибки в базу данных и их можно посмотреть через интерфейс системы. Меню **System -> Лог ошибок**.

10.2.5.8.2. По умолчанию система сохраняет ответы методов вида MyError, а также все console.error выводимые в процессе работы сервера. Также когда на сервере где-то происходит не пойманная ошибка (uncaughtException), она ловится на самом верхнем уровне и также логируется.

10.2.5.8.2.1. Исключением являются ошибки связанные с доступом к БД.

10.2.5.8.3. Настройка системы логирование в БД

10.2.5.8.3.1. Вы можете настроить, что именно будет логироваться в базу через интерфейс системы. Более того, эти настройки будут применены на лету, то есть вам не потребуется перезапускать сервер, чтобы изменения вступили в силу.

10.2.5.8.3.2. Настройки делаются через таблицу **System -> Системные настройки** (вы можете встретить устаревшее название Клиентские настройки).

10.2.5.8.3.2.1. Нужно изменить или создать (она может быть не создана) строку с Системным именем “**LOG_ERRORS**”.

10.2.5.8.3.2.2. В “Значении 1” перечислите через запятую типы логируемых ошибок из этого списка: ANY|NO|MYERROR|USERERROR|UNKNOWN. Если среди перечисленных будет значение NO, то никакие ошибки не будут логироваться. Если есть ANY (но нет NO), то любые типы ошибок будут логироваться.

10.2.5.8.3.2.3. В “Значении 2” перечислите через запятую источники логируемых ошибок из этого списка: ANY|NO|API|LOG|THROW. Значение NO и ANY работают также как и для типов. API - если ошибка в ответе от метода вызванного через внутренние API, LOG - ошибка выведенная через console.error, THROW это не пойманные ошибки.

10.2.5.8.3.3. Как указано выше, вы можете настраивать типы ошибок и источники ошибок. По умолчанию используются ANY и ANY.

10.2.5.9. Логирование с клиента по сокету.

10.2.5.9.1. Система поддерживает возможность писать в серверный лог сообщения отправленные с клиента по сокету в канал “logFromClient”.

socket.on('logFromClient', …)

10.2.5.9.2. Этот механизм особенно полезен для отладки виджетов или мобильных приложений, когда нет возможности смотреть логи на стороне клиента, важные детали можно залогировать таким образом через сервер.

10.2.5.9.3. Такой лог выделяется в блок:

logFromClient START 1 2025.07.11 13:40:18
{ msg: 'TEST', abc: 489 }
logFromClient END 1

10.2.5.9.3.1. После слова START идет ID пользователя, для которого логируется.

10.2.5.9.4. По умолчанию, сервер игнорирует такие отправки, но вы можете включить логирование в настройках и указать пользователей, от которых принимать такие сообщения.

10.2.5.9.5. Конфигурация “log:socket:”

10.2.5.9.5.1. **logFromClient**. Включает или отключает логирование.

10.2.5.9.5.2. **logFromClientUsers**.Массив с ID пользователей, для которых логировать. Можно указать “-” чтобы логировать от неавторизованных.

10.2.5.10. Лог отказов доступа

10.2.5.10.1. Как описано в разделе про доступы, отказы логируются в Доступ -> Отказы доступа.

10.2.5.11. Лог отказы авторизации

10.2.5.11.1. Аналогично отказам в доступе, также логируются неудачные попытки авторизации). Доступ -> Отказы авторизации.

10.2.5.11.2. Логируются поля: user_id, login, sid, user_agent, address, err.

10.2.5.11.2.1. Введенный пароль естественно не логируется

10.2.6. Просмотр логов на сервере

10.2.6.1. Как правило на сервере процесс развернут в контейнере и смотреть логи можно через стандартный журнал с фильтрацией по контейнеру.

10.2.6.1.1. Пример: journalctl --since "2025-07-03 07:00:00" --until "2025-07-03 11:00:00" CONTAINER_NAME=app | grep -C 2 "MyError".

10.2.6.1.2. Если вы разворачивали сервер по инструкции в этой документации, то вы можете набрать ./start.sh log ccs_app, где ccs_app - имя сервиса.

10.3. Клиентские логи

10.3.1. Клиентская приложение выводит логи в консоль в соответствии с клиентским кодом, однако есть логи взаимодействия с сервером, которыми можно управлять. Речь идет о клиентах, где для связи используется наша библиотека go_core_query (см в npm). См. Подключение к бэкэнду.

10.3.2. Выводить ли логи взаимодействия с сервером определяется при инициализации go_core_query. Как правило, в наших реализациях признак берется из куки debugMode, которая переключается функцией debug(). Но по факту, это параметр debug:boolean при инициализации нового инстанса подключения.

10.3.2.1. Если флаг включен, то в консоль будет выводиться две группы. Первая, светло-салатовая, выводит только ушедший запрос, а вторая, темно-зеленая, содержит уже два объекта: запрос и ответ. При такой реализации объект запроса в консоли выводится дважды, но это сделано специально, так как между запросом и ответом могут быть и другие запросы и чтобы видеть полную картину, мы выводим информацию и когда запрос ушел и когда он вернулся.

10.3.3. В параметрах инициализации инстанса подключения имеется также параметр **debugFull**, и если его указать true, библиотека выводит значительно больше информации о своих действиях, что позволяет разобраться с проблемой, если вам не удается установить соединение с сервером или в еще каких-либо нештатных ситуациях.

10.3.4. Глобальные объекты клиентской части GoCore

10.3.4.1. Фронтенд системы написан давно но основательно. Он имеет системы окон и таблиц, выпадающих списков и других компонентов. Часть информации, о загруженных инстансах и настройках, хранится в глобальном объекте **MB**. Там содержится очень много информации, вы можете при желании сами изучить этот объект. Однако вот что вам может пригодиться: **MB.Tables.tables**, **MB.Forms.forms**, **MB.Frames.frames**. Это массивы с инстансами, где содержится вся загруженная информация (о профайле, о полях, сами данные, все настройки, состояния).

10.3.4.2. Также вы можете получить информацию о пользователе через MB.User.

11. Роутинг

11.1. Роутинг в системе работает на основе Express и обработка производится в файлах /routes/, начиная с index.ts

11.2. **Важно!** На серверах имеется промежуточное звено nginx, который обрабатывает все http(s) запросы, кроме добавленных в исключения (см. Настройте параметры -> ccs_app_nginx_location)

11.3. В системе уже описаны необходимые для работы роуты, включая API.

11.4. API

11.4.1. Роут выглядит так: /api/v1/:className/:command. Обрабатывается только POST запросы.

11.5. upload/files

11.5.1. Система обрабатывает загрузку файлов, в том числе и в не публичную зону, а также их скачивание при наличии доступа.

11.6. При добавлении новых роутов, добавляйте их в конце файла, а логику выносите в отдельные файлы в той же директории.

11.7. Middleware

11.7.1. Crossorigin

11.7.1.1. Этот мидлвере позволяет фильтровать кросс ориджин запросы. Список доменов, которым разрешен доступ определяется через интерфейс, меню **Settings -> Origin**.

11.7.2. loadUser

11.7.2.1. Позволяет загрузить сессию пользователя. Обязателен для авторизированных запросов.

12. Блокировки

12.1. Принцип работы

12.1.1. Система имеет механизм блокировки записи или всей функции. Данные блокировок хранятся в определенном объекте в памяти процесса, однако все подготовлено для того, чтобы вынести этот объект, например в Redis, чтобы обеспечить возможность запуска нескольких инстансов системы с единой информацией о блокировках.

12.2. Блокировка записи

12.2.1. Разработчик может повесить блокировку на конкретную запись определенной сущности, тогда при попытках ее изменить или удалить, система будет искать ключ блокировки в объекте запроса, если не найдет, то в родительском запросе и так до верха.

12.2.1.1. На методы изменения и удаления блокировка ставится автоматически. Разработчик же может выполнить блокировку записи еще в начале кастомного метода, что позволяет делать реализовывать различную логику, например проверки, не опасаясь, что пока они ведутся, исходные данные могут быть изменены.

12.2.2. Если ключ найден, то операция допускается. Поиск в родительских объектах запроса позволяет выполнять манипуляции с записью дочерним методам, что позволяет свободно выносить логику в отдельные методы и самому не заниматься передачей в них ключа блокировки.

12.2.3. Если нет (какой-то другой метод пытается изменить ту же запись), то есть два режима:

12.2.3.1. В первом (по умолчанию), запрос с небольшой периодичностью пытается повторно провести блокировку (блокировка вызывается из кастомного метода, если разработчик в нем ее написал, или непосредственно в операции изменения или удаления автоматически системой.

12.2.3.1.1. Попытки продолжаются, пока запись не освободится или не пройдет выделенное на попытки время. По умолчанию это 2 минуты, но при вызова блокировки,

разработчик может задать другое время параметром “**lockTime**” (в миллисекундах).

12.2.3.2. Во втором, если при вызове блокировки (если разработчик сам ее вызывает) передать параметр “**reject**”=true, то в случае если запись занята другой блокировкой, система сразу отклонит запрос, без повторных попыток с небольшим интервалом.

12.2.3.3. Вызов блокировки выглядит так:

```
const res = await lock(r, 'order_', id)
if (res.code) {
   return new MyError('Не удалось заблокировать заказ для Order_.calcAmounts')
}
```

12.2.3.3.1. Объект запроса, имя сущности, ID записи которую блокировать и, четвертым необязательным параметром, объект с параметрами блокировки.

12.3. Блокировка функции

12.3.1. Иногда нужно заблокировать сразу весь метод, чтобы он не вызвался повторно, пока не будет завершен текущий вызов. Это делается также как и с блокировкой записи, первый параметр также объект запроса, второй параметр может быть любой строкой, например имя функции, третий null, а четвертый, необязательный объект с параметрами.

```
res = await lock(r, this.name, 'syncBJ', null, {reject:true})
if (res.code) {
   return new UserOk('Предыдущая итерация syncBJ еще выполняется. Пропустим этот запуск.')
}
```

12.4. Снятие блокировки

12.4.1. Блокировка снимается автоматически, когда завершается метод, в котором она была установлена. Это происходит в модуле **api**, через который происходит вызов любого метода.

12.4.2. Разработчик также может самостоятельно снять блокировку вызвав unlock

```
unlock(name:string, id:number|string, key:string): IError
```

12.5. Мониторинг и управление

12.5.1. На практике, мне не разу не приходилось пользоваться этими методами, тем не менее упомяну. Есть два метода **showLocks** и **unlockRec**, которые позволяют просмотреть список текущих блокировок и снять любую блокировку. Методы относятся к CoreClass, то есть имеются у любого класса.

12.5.1.1. Метод unlockRec не позволяет снять блокировку всей функции. Если понадобится такой метод, его можно будет доработать.

13. Система кэширования

13.1. Механизм кэширования построен в системе таким образом, что запрашивается только то, чего еще нет в кэше, вплоть до колонок, полученные данные пополняют кэш, а система очистки кэша, при изменении данных все зависимые данные из всех зависимых сущностей. Это достигается благодаря тому, что система досконально знает все связи между данными опираясь на профайлы.

13.2. Механизм работает автономно и не требует вмешательства разработчика, однако есть опции, которые могут пригодиться.

13.2.1. Вы можете полностью отключить механизм, указав в конфиге “**useCache**”:false.

13.2.2. Вы можете отключить использования кэширования для определенного запроса передав в параметры запроса “**use_cache**”=false (так сложилось, что в одном случае camelCase, а в другом snake_case).

13.2.2.1. Это параметр относится именно к запросу, а не к параметрам метода, то есть система его получает из **r.rParams**, а не из **r.data.params**, однако вы можете его передавать в параметрах метода, он автоматически будет перенесен в перенесен куда надо.

13.2.3. Кэш в настоящее время хранится в подготовленном объекте в памяти основного процесса, однако он подготовлен к тому, чтобы быть вынесенным в отдельный сервис, например Redis, для возможности запуска нескольких инстансов с единым кэш.

13.2.4. Еще не до конца реализовано

13.2.4.1. На уровне класса, после загрузки профайла, вы можете определить (например в переопределенном методе **init**) поле **cache_max_length** (в профайл: this.class_profile.cache_max_length) Оно определит максимальное кол-во строк может хранится в кэше для этого класса. Параметр есть, а механизм очистки сверх лимита (самых старых в кэше) еще не реализован. Также при доработке имеет смысл вынести cache_max_length в профайл в СУБД, то есть расширить структуру класса class_profile

13.2.4.2. В настоящее время система не контролирует переполнение памяти. Нужно будет дописать контроль максимального указанного в конфигурации лимита по памяти и вычищать при необходимости более старые данные.

13.2.5.

14. Механизм миграции базы между разработчиками.

14.1.

15. Механизм сессий

15.1. Принцип работы

15.1.1. Подключение к бэкэнду осуществляется через сокет или при http(s) запросе. При этом система берет токен из заголовка авторизации (Authorization header: Authorization: Bearer <Token>) и загружает сессию.

15.1.2. Для каждого устройства своя независимая сессия. При этом для нескольких вкладок одного браузера сессия будет одна с соответствующим количеством клиентов. Для другого браузера будет новая сессия.

15.1.3. Если пользователь ранее не авторизировался или если токен просрочен (время жизни самого JWT закончилось), то генерируется новый токен и записывается в заголовок авторизации.

15.1.3.1. Если время жизни самого токена закончилась, но сам токен валиден, то в новый токен переносятся данные сессии из старого токена.

15.1.4. После получения или генерации токена, система загружает сессию и данные пользователя (если в базе есть активная сессия) из базы.

15.1.4.1. Если сессии нет в базе, то сессия будет анонимной, а все запросы (кроме запроса авторизации) будут возвращать стандартный ответ (IAPIResponse) с кодом -4. При этом в базу такая сессия не пишется.

15.1.4.2. Сессия в базе создается при успешной авторизации пользователя, а после обновляются данные анонимной сессии, которая перестает быть анонимной.

15.1.5. Загруженные данные сессии (или данные анонимной сессии) записываются в socket.handshake. Записывается сама сессия в

socket.handshake.session и данные клиента в socket.handshake.currentClient.

15.1.5.1. Напомню, сессия может содержать несколько клиентов (несколько вкладок браузера). Хранятся в объекте session.clients (см. ISession).

15.1.6. Данные загруженной сессии доступны в объекте запроса IAPIRequest (**r.client и r.client.session**). Через этот же объект вы можете получить других клиентов (остальные вкладки) той же сессии (**r.client.session.clients**).

15.1.6.1. Для подключения по сокету, данные формируются при его подключении и, позже, используются во время запросов, так как после подключения эта процедура не повторяется.

15.1.6.2. Для http(s) запросов сформированные данные используются сразу в запросе.

15.1.6.3. В данных клиента уже загружена информация о пользователе, разработчик может его получить из объекта запроса: **r.client._session.usr** (IUser). В этом же объекте есть данные записи пользователя в БД: **r.client._session.usr.userData** (IUserDataRow). Наиболее распространенное использование это получение ID пользователя: **r.client._session.usr.userData.id**.

15.1.7. При запросе система проверяет авторизован ли пользователь и не истек ли период жизни сессии. В случае если срок действия сессии прошел или пользователь еще не авторизовался, запрос не проходит дальше, а возвращает стандартный ответ с кодом -4 (уже упоминалось выше), а также всем клиентам сессии (если это подключение по сокету), эмитится событие “logout”. По этому событию во всех вкладках осуществляется переход на страницу авторизации.

15.1.7.1. Это относится к встроенному фронтенду системы. Для кастомных систем, если пользователь использует библиотеку go_core_query, то вызывается метод authFunction, определенный при инициализации (если указан параметр autoAuth = true, то сразу будет произведена авторизация, а authFunction вызван не будет). Для систем не использующих библиотеку, но подключающийся по сокету, разработчик может самостоятельно обработать это событие.

15.1.8. Пролонгация сессии

15.1.8.1. В фоновой задаче продлевается срок действия сессии, если по ней последний запрос был не ранее, чем время жизни сессии.

15.1.8.1.1. Если до окончания срока действия сессии еще далеко (больше чем период вызова фоновой задачи * 3), то продления не производится, чтобы не нагружать сервер.

15.1.9. Отключенные сокеты удаляются из памяти в фоновой задаче, если те не переподключились в течении долгого времени.

15.2. Ограничения сессий

15.2.1. Срок действия сессии (Session lifetime).

15.2.1.1. При создании сессии, ей определяется срок ее жизни опираясь на тип устройства и данные из конфигурации.

15.2.1.2. Тип устройства, передается при подключении в “query” в поле “device_type” и может содержать следующие значения: ‘WEB’ (для браузеров), 'APP' (для мобильных приложений), 'API' (для внешних сервисов), 'SYS' (системная сессия), 'BY_USR' (для внутреннего административного механизма подключения за другого пользователя - используется в тестировании). Эти типы носят информационный характер и не в коем случае не влияют на доступ, так как он может быть задан произвольно. Однако для каждого типа можно указать свое время сессии в конфигурационном файле (**session:expiredMin**).

15.2.1.2.1. Планируется немного доработать систему и позволить определять ограничения на уровне пользователя, но это будет сделано при необходимости.

15.2.2. Количество запросов в минуту

15.2.2.1. Система отслеживает количество запросов в минуту от одной сессии и если это число превышает заданное, то запросы отклоняется.

15.2.3. Количество активных клиентов одной сессии (кол-во вкладок)

15.2.3.1. Система отслеживает количество одновременно активных клиентов для одной сессии (как правило это несколько

вкладок), и если оно превышает заданное, то сокет не подключится.

15.2.4. Блокировка по IP

15.2.4.1. Система отслеживает, с какого ip приходит запрос или подключение и если он содержится в списке на блокировку, запрос отклоняется. Пока список можно разместить только в конфиге (**session:blocked_ips**) (требует перезапуска сервера), однако это можно доработать при необходимости.

15.3. Управление сессиями

15.3.1. Администраторы системы могут смотреть и управлять активными сессиями через интерфейс системы. **Меню Access->User session**.

15.3.2. Администратор может просматривать и менять параметры сессии, такие как кол-во запросов в минуту, кол-во активных вкладок, время жизни сессии с момента авторизации или последнего запроса (см. пролонгация), последний запрос (timestamp), userAgent. Также есть поле expired (timestamp), которое при желании тоже можно изменить.

15.3.2.1. Также администратор может удалить сессию, тогда клиент не сможет больше делать запросы, а все его открытые вкладки с системой будут перенаправлены на страницу авторизации (если это фронтенд системы/используется go_core_query).

15.4. Механизм user_mode

15.4.1. Задуман как возможность переключаться между ролями одного и того же пользователя, например он может пользоваться системой как, например, менеджер, но также и как клиент организации. Чтобы не ему не заводить двух пользователей, а просто переключаться между режимами был разработан данный механизм.

15.4.2. В системе для него подготовлены все необходимые функции, однако для конкретного проекта необходимо будет создать режимы, определить им набор доступов, указать сущность, например “organization”.

15.4.3. При подключении пользователя к, например, менеджерам организации (Org_manager_user.add), необходимо будет вызывать метод User.createUser (если еще не создан), передавая указания режима и ID в сущности “org_manager”. Тогда для пользователя будет создан дочерний пользователь с режимом MANAGER для организации с определенным ID. Если нужно подключить

пользователя в качестве менеджера еще к одной организации, то у него появится еще один дочерний пользователь.

15.4.3.1. При отключении пользователя от организации (в роли менеджера), достаточно заблокировать соответствующего дочернего пользователя.

15.4.4. Когда у пользователя появляется несколько режимов (дочерних пользователей), он может переключаться между ними используя методы getModes и changeMode.

15.4.5. За более подробной информацией обращайтесь в техподдержку.

16. Система доступов

16.1. Система доступов разделена на два механизма, это проверка доступа пользователя к методу (в простых проектах этого достаточно), и проверка доступа к данным.

16.2. Описание процесса

16.2.1. Система включается только для пользователей типа USER, а для ADMIN проверки пропускаются. Также доступны проверяются только для внешних запросов (с клиента или по API).

16.2.2. Доступы загружаются при инициализации сессии и далее хранятся в ней. Обновляются при изменении.

16.2.3. Есть некоторые настройки в конфигурации внутреннего api, они расположены в libs/api/apiConfig. Там имеется файл **default.ts** и **project.ts**. В первом располагаются настройки ядра, а во втором они могут быть дополнены настройками для конкретного проекта. Настройки из этих двух файлов объединяются.

16.3. К операциям

16.3.1. При поступлении запроса, система проверят (исключая методы не требующие проверки, определены в конфигурации libs/api/apiConfig, как doNotCheckCommands и схожие), есть ли хоть у одной роли подключенной к пользователю разрешение к этой операции, а также нет ли у него роли, которая строго запрещает такой доступ. Запрет приоритетнее.

16.3.1.1. Если нет явного разрешения, то доступ запрещен. То есть хоть у одной роли должно быть разрешение и при этом ни у одной не должно быть запрета.

16.3.1.2. Доступ может быть разрешен или запрещен для всех методов класса.

16.4. К данным

16.4.1. Доступ к данным разделяется на два отдельных механизма. Первый, это ограничение на получение данных, то есть на все “get” (SELECT) запросы, в том числе и внутренние, а второй ограничения на все прочие методы, как базовые, так и кастомные.

16.4.1.1. Ограничения на получение данных (метод “get”)

16.4.1.1.1. Проверка доступа к данным работает на основе списка доступа и применяется при каждом запросе, в том числе и при внутренних запросах. То есть в отличии от проверки доступа к операциям, которые проверяются только на входе извне, система доступа к данным накладывает ограничения при запросе данных из базы в методе “get”.

16.4.1.1.1.1. Такая проверка, естественно, выполняется только если для данного класса и метода включена настройка доступа по списку.

16.4.1.1.2. Важной, и не совсем очевидной особенностью данного механизма, является как раз ограничения доступа к данным для внутренних запросов, то есть таких, когда изначальный запрос проходит проверку доступа (не важно ограниченного списком или нет), а дальше внутри этого метода выполняются подзапросы других данных, и если для этих данных есть настройка доступа по списку, то она будет применена. Рассмотрим на примере вымышленного метода Product.getAvailable, который должен вернуть доступные для приобретения товары. Допустим, что такой метод сперва получает активные Категории, а еще, допустим, что определенным категориям клиентов, мы ограничиваем набор Категорий товаров, выдавая как раз доступ по списку (список корректируется автоматически в ходе отдельных бизнес-процессов). Тогда в ходе метода, при получении Категорий, сработает механизм доступа по списку и метод получит ограниченный набор Категорий, для которых уже будут собраны доступные Продукты. В приведенном примере, должен быть настроен доступ к методу “get” класса “Product_category” с признаком “Access_by_list”.

16.4.1.1.3. Этот механизм гарантирует, что если пользователю ограничен набор данных, то он не сможет их

получить из за невнимательности разработчика при реализации сложных алгоритмов.

16.4.1.1.3.1. При необходимости, разработчик может обойти эти ограничения во внутренних методах, передав параметр **doNotCheckList**. В этом случае разработчик должен отдавать себе отчет зачем он это делает и как это скажется на бизнес-процессах.

16.4.1.2. Ограничение доступа для определенных данных для прочих методов

16.4.1.2.1. Не только получение данных может быть ограничено списком, но и прочие методы, как базовые (modify/remove/…), так и кастомные (например описанный выше getAvailable).

16.4.1.2.2. Это позволяет гибко настраивать возможности пользователя по взаимодействию с данными, например, просматривать можешь все Заявки своей организации, а обрабатывать (Принимать, Отклонять, Выполнять), только те, которые соответствуют твоей специальности (следовательно к которым выдан доступ для соответствующих операций, например setAccepted).

16.4.1.2.3. Механизм применим к операциям работающим по определенному id или ids и не может быть применен к операциям с неопределенным набором данных, например метод (вымышленный) applyAll. В этом операция пропускается. Однако внутри методов с неопределенным набором данных, как правило внутри есть запросы на получение уже определенных наборов, которые фильтруются по списку, при наличии такой настройки для роли пользователя для запрашиваемого класса.

16.4.1.2.3.1. Для операций работающих по ids (исключая операцию “get”) имеется два режима ограничения, в случае если хотя бы один id не проходит проверку. В первом случае (по умолчанию) идет отказ доступа для операции, во втором, не найденные id исключаются из параметров. Настройка называется access_by_list_filter_only. Ее имеет смысл включать (сделать так чтобы отказа не было,

а просто фильтровался входной список) для кастомных операций типа “get”.

16.4.1.2.4. Часто проще всего (если это соответствует бизнес-логике) настроить доступ таким образом, чтобы доступ к операции, например “modify”, допускался для того же списка, что и операция получения данных (“get”). Это можно сделать установив для указанной операции, (например “modify”) галочки **is_access_by_list** и **Is_access_by_list_like_get**.

16.4.1.2.4.1. Также можно сделать чтобы список наследовался от списка для “get”, но вдобавок применялся собственный список, для этого установите **is_access_by_list_like_get_plus_self**.

16.4.2. Иерархическое распространение доступа из списка

16.4.2.1. Список доступа представляет собой таблицу связи операции класса, конкретной записи в этом классе и пользователя или роли, кому предоставляется доступ к этой записи.

16.4.2.2. Разумеется предоставлять доступ к разветвленной структуре данных довольно накладно, даже если это делается автоматически, поэтому система предусматривает возможность наследования доступа как для дочерних записей той же сущности, если сущность иерархическая, так и к дочерним сущностям. Конфигурацию такого доступа можно очень гибко настраивать.

16.4.2.2.1. На примере, мы можем предоставить пользователю доступ к определенному Проекту и указать, что доступ будет распространяться на все его дочерние Подпроекты, а также на Изображения_проекта (каждого, к кому наследуется доступ), Характеристикам_проекта и прочим дочерним сущностям (опционально). В свою очередь можно также отнаследовать доступ к Авторам_изображений_проекта, например, и так далее.

16.4.2.2.2. Настройки доступа к операциям с признаком “Доступ по списку” и структура наследования доступа регулируется на уровне Роли, которая в дальнейшем выдается пользователю.

16.4.2.3. Автоматическое управление списком

16.4.2.3.1. Чтобы выдать доступ к конкретной записи, необходимо добавить запись в таблицу связи. Правильнее всего это делать автоматически, например переопределив и дополнив метод добавления записи к сущности определяющий отношение пользователей к определенной сущности. На примере Проектов, эта сущность может быть Участники_проекта или, например, Администраторы проекта. То есть по изменению состава Участников, может вызываться метод синхронизации (его нужно написать, но это довольно просто) пользователей в этом списке и Списка_доступа к нужным операциям. Для Администраторов, например будут добавляться записи для операций get/add/modify/remove в список доступа для этих пользователей, а для Участников_проекта, только для операций get/modify. При этом разумеется у Администраторам должна выдаваться роль “Администратор проекта”, для которой будет настроен доступ к этим операциям с галочкой “доступ по списку”, и соответственно Участникам, своя роль, со своими настройками. Таким образом, метод синхронизации Корректирует для пользователя набор ролей, и список доступа.

16.4.2.3.2. Подключение Участников или Администраторов или еще какой-либо связи реализуется через интерфейс системы, например с помощью TwoColumnEditor.

16.5. Управление доступом

16.5.1. Выдача ролей пользователю

16.5.1.1. Роли пользователю могут быть выданы вручную или автоматически, программным кодом, как описано в примере выше (Автоматическое управление списком).

16.5.1.2. Для автоматической синхронизации существует метод User_role.sync. Он добавит недостающие из указанного списка и уберет не указанные. Также будет добавлена роль BASE_ACCESS, необходимая для любого пользователя, кроме администраторов.

16.5.1.2.1. Однако это приемлемо, если у вас только один источник выдачи ролей, а на практике роли могут

быть выданы из нескольких источников, а также какие-то роли могут быть выданы вручную. Поэтому, при сложной логике выдачи ролей, следует написать отдельный метод, который будет собирать информацию со всех источников, и синхронизировать на основе всех данных, а роли не относящиеся к каким-либо источникам затрагивать не будет.

16.5.1.3. Вручную можно выдать роль пользователю через таблицу пользователей, через пункт контекстного меню “Роли пользователей”.

16.5.1.4. Несколько ролей накладываются друг на друга. То есть пользователь получит доступ к тому, что разрешено хотя бы одной из его ролей, при этом не запрещено ни одной из его ролей.

16.5.2. Настройка роли

16.5.2.1. Прежде всего сама настройка сохраняется в таблице Доступ->Доступ к операциям. Там вы можете добавить доступ к определенной операции, для определенной роли и указать особые настройки.

16.5.2.2. Список операций располагается в таблице Доступ-Операции класса. Чтобы методы новых классов там появились, а также чтобы у существующих классов добавились там вновь созданные методы, необходимо провести синхронизацию (когда разработали что-то новое). Это делается в этой таблице, в контекстном меню “Refresh list”.

16.5.2.3. В настоящий момент нет хорошо проработанного интерфейса для наглядного управления доступами для каждой роли. Однако пару механизмов, упрощающих настройку.

16.5.2.3.1. Прежде всего вы можете просто добавлять записи в таблицу Доступ к операциям.

16.5.2.3.1.1. Там необходимо выбрать операцию, роль и установить галочку is_access (и другие настройки при необходимости).

16.5.2.3.1.2. Вы можете добавить доступ ко всем операциям класса, выбрав операцию “*” этого класса.

16.5.2.3.1.3. Все настройки описаны ниже.

16.5.2.3.2. Выдача доступа по отказам.

16.5.2.3.2.1. Отказы доступа логируются в таблице Доступ->Отказы доступа.

16.5.2.3.2.2. Когда происходит отказ, в таблице появляется запись с отказом этой операции, а через контекстное меню “Выдать доступ”, можно выдать доступ Текущей настраиваемой роли.

16.5.2.3.2.2.1. Через это же контекстное меню можно забрать доступ.

16.5.2.3.2.3. Чтобы выбрать, какую роль настраивать, вам необходимо зайти в таблицу Доступ->Роли и установить галочку “Выставлять доступ”, при этом сняв ее у всех других ролей.

16.5.2.3.2.4. Алгоритм действий

16.5.2.3.2.4.1. Выбрать роль для настройки, создать тестового пользователя и выдать ему роль BASE_ACCESS и ту, которую будете настраивать. Выдать доступ к нужным меню (см. отдельный пункт). Зайти под этим пользователем в систему в отдельном браузере (например в режиме Инкогнито), планомерно выбирать нужные пункты меню и весь необходимый функционал, получая отказы, и выдавая доступ через таблицу отказов.

16.5.2.3.2.4.1.1. Так как после каждого отказа переходить к отказам (в соседнем браузере) и выдавать по одному довольно утомительно, есть специальный режим позволяющий для конкретного пользователя не получать отказы, а просто их логировать. То есть отказ зафиксирован, но операция успешно выполняется. Таким образом можно массово пройти большой блок функционала, а

потом массово выдать доступ по всем отказам из таблицы.

16.5.2.3.2.4.1.1.1. Чтобы активировать режим, необходимо прописать id этого пользователя в конфигурации config/config.json в параметр enableSkipNoAccessForUsers (добавить id в массив), после чего перезапустить основной процесс.

16.5.2.3.3. Доступ к меню

16.5.2.3.3.1. Чтобы выдать доступ к меню, вам необходимо выбрать роль для настройки - галочка “Выставлять доступ” в таблице ролей. Далее зайти в таблицу System->Редактор меню, найти нужные пункты и через контекстное меню Выдать доступ.

16.5.2.4. Настройка доступа с наследованием. Иерархический доступ по списку.

16.5.2.4.1. В описании “Иерархическое распространение доступа из списка” мы говорили о том, что выданный пользователю доступ к конкретной операции и к конкретной записи (таблица Список доступа) может наследоваться как на дочерние записи этого же класса, так и на дочерние сущности. Но чтобы это работало, для роли должны быть соответственно настроены доступы.

16.5.2.4.2. Чтобы доступ к операции был по списку, то есть только к тем записям которые обозначены в списке (возможно с наследованием), у записи выдающей доступ определенной роли, например “project.get PROJECT_ADMIN” (метод “get”, класса “project”, для роли “PROJECT_ADMIN”), необходимо указать галочку is_access и галочку is_access_by_list. Чтобы доступ распространялся на все подпроекты, указанных в списке проектов, нужно добавить галочку “self_hierarchical”.

16.5.2.4.3. Чтобы доступ распространялся на дочерние сущности, необходимо для этих сущностей добавить новые записи в Access_to_operation. Например, чтобы дать доступ к Изображением тех проектов, к которым предоставлен доступ, надо добавить запись доступа для той же роли: “project_picture.get PROJECT_ADMIN”, установить галочку is_access и галочку is_access_by_list, а далее необходимо указать информацию о родительском доступе: Выбрать родительскую операцию (поле Parent name) “project.get PROJECT_ADMIN” и указать поле parent_foreign_key (это поле, которое в таблице project_picture указывает на таблицу project и вероятнее всего это будет “project_id”).

16.5.2.4.3.1. Если изображения имеют иерархию, то вы также можете задать иерархический доступ, при желании.

17. Документирование

17.1. В системе имеется механизм автоматического сбора документации в формате openAPI. Она выводится в виде HTML-страницы с помощью Swagger UI по адресу **/api-docs**.

17.1.1. Вы также можете скачать файл спецификации по URL **/api-docs/download** и использовать его во внешних инструментах.

17.1.2. Большинство методов требуют авторизации, поэтому при использовании документации сперва выполните метод **User.login**, а полученный JWT укажите в секции авторизации (кнопка **Authorize** в правом верхнем углу страницы).

17.2. Документация формируется для методов классов на основе дженериков входных и выходных параметров. Также в документацию попадает документирующий комментарий. Типы и интерфейсы, за исключением базовых типов, попадают в раздел схем документации.

17.2.1. Чтобы метод попал в документацию, необходимо обозначить его декоратором **@OpenApi()** непосредственно над функцией (после документирующего комментария).

17.2.2. Если параметр в дженерике будет анонимным, то есть описание объекта в фигурных скобках, то все используемые внутри собственные типы и интерфейсы будут полностью раскрыты в итоговой документации, что иногда может быть не очень удобно

для чтения. Если хотите чтобы там осталось имя типа/интерфейса, создайте его и подставьте в дженерик.

17.2.2.1. Пример с анонимным типом/интерфейсом:

```
@OpenApi()
async create(r: IAPIRequest<{
   orders: ICreatePmntOrder[]
   for_user_id?: number
}>)
```

17.2.2.2. Пример с вынесенным типом/интерфейсом:

```
export interface ICreatePaymentRequest {
   orders: ICreatePmntOrder[]
   for_user_id?: number
}
```

```
@OpenApi()
async create(r: IAPIRequest<ICreatePaymentRequest>):
```

17.3. Сборка документации осуществляется командой “**node ./openAPI/openApi.js**”, в package.json имеется соответствующий скрипт.

17.3.1. Итоговый файл располагается в openAPI/swagger.yaml. Он каждый раз пересоздается, поэтому его не следует править руками.

17.4. Пожалуйста, пишите достаточный документирующий комментарий для функций. Также пишите комментарии к параметрам, если их поведение не является очевидным или не описано выше. Если вы пишите какой-нибудь модуль, желательно в начале файла расписать логику работы и граничные условия. Если по проекту ведется отдельная документация, поддерживайте ее в актуальном состоянии по согласованию с руководителем проекта/тимлидом.

18. Динамические поля

18.1. Концепция

18.1.1. Механизм динамических полей позволяет работать со сложными структурами данных так, как если бы это была плоская таблица. А благодаря существующим в системе механизмам получения и редактирования данных, имеется возможность работать в том числе и с мультизначениями для одной записи в основной плоской таблице.

18.1.2. Как это работает

18.1.2.1. В минимальной конфигурации в процессе участвуют 3 сущности. Основная сущность, та которая формирует плоские данные. Вторая сущность, эта та, которая преобразуется в поля основной сущности, то есть это те столбцы, которые должны появиться в основной таблице, чтобы данные более полные. Третья сущность, это таблица

связи, таблица со значениями второй сущности для записей в первой.

18.1.2.2. Как правило это набор характеристик, какой либо сущности. Эти характеристики могут быть разных типов. Тип характеристики определяет тип редактора, который будет установлен (может быть переопределен) в конечной таблице или форме.

18.1.2.3. Значений характеристики для конкретной записи сущности также может быть несколько, механизм поддерживает возможность работы с мультизначениями. Клиентские таблицы умеют отображать такие данные и позволяют кастомизировать редактор (см. Управление -> Фрейм для редактора).

18.1.2.4. Рассмотрим на примере Товаров и их Характеристик. Может быть множество характеристик для различных товаров и так как они могут добавляться и удаляться в процессе работы с системой, а также в силу их большого количества, они не могут быть физическими полями в основной сущности (Товар), кроме того такая структура не позволила бы иметь несколько значений для одной Характеристики конкретного Товара. Поэтому структура может быть следующей. Таблица Товаров, Справочник Характеристик и таблица со Значения_Характеристики_Для_Товара.

18.1.2.4.1. Таблица со значениями может быть более сложной структурой, так как, например, для значений вида Справочник, потребуются отдельные таблицы с их значениями, но в этом случае имеет смысл средствами SQL привести таблицу со значениями в уже подготовленную “VIEW”, где хранятся в том числе и физические значения.

18.1.2.5. Итак, у нас есть Товары и их Характерики (значения). Если мы хотим вывести данные для одного Товара, то проблем особых не возникает, так как мы можем получить необходимые данные в 2-3 запроса. Однако если мы хотим работать с таблицей, пусть даже с пагинацией, то требуется другой подход получения данных (особенно если вы хотим выгрузить в Excel несколько тысяч записей). Механизм строит оптимальный запрос для получения таких данных, выводя в результате все характеристики как если бы это были поля этой таблицы. А благодаря профайлам, система знает, что это поля из другой таблицы и знает как с ними работать, чтобы обеспечить редактирование.

18.1.2.6. Ограничение набора динамических полей

18.1.2.6.1. Как правило в подобных структурах данных, характеристики относятся к определенным категориям сущности (например Категории_Товаров), тогда нам необходимо иметь возможность ограничить набор динамических полей определенной категорией (и ее дочерними). Система позволяет это сделать, указав таблицу для фильтра. Тогда если данные загружаются в форме (например форма определенной Категории), то система обратится в таблицу фильтров и наложит ограничения таким образом, что будут подгружены только характеристики относящиеся к этой категории.

18.1.2.6.1.1. Иерархия по таблице фильтрации.

18.1.2.6.1.1.1. Если таблица, по которой осуществляется фильтрация является иерархической, как в нашем примере с Категориями (у Категории могут быть подкатегории, а у них свои, и так далее), то фильтр может это учитывать (при определенной настройке) и выдавать Характеристики не только для указанной Категории, но и всех ее дочерних и/или всех ролдительских.

18.1.2.6.2. Справочник характеристик также может быть по умолчанию отфильтрован по какому-либо признаку, не обязательно чтобы все Характеристики попадали в динамику.

18.1.2.6.3. Подытоживая, механизм гибкий и может быть доработан еще под большую кастомизацию, если позволить определять функции, которые будут подготавливать срезы данных опираясь на входные параметры. Однако текущей реализации достаточно для описанного сценария. Он позволяет простой настройкой реализовать сложный механизм взаимодействия с данными.

18.1.2.6.4. Наследование значений

18.1.2.6.4.1. Ядро поддерживает механизм наследования значений для обычных физических полей в

иерархических сущностях. Для этого служит настройка is_inherit в профайле поля. Для динамических полей имеются собственные настройки профайла, которые динамически дополняют профайл основной сущности (см. Управление -> Настройки профайла динамических полей). Там имеется, в том числе, настройка is_inherit, которая, как и для физических полей, позволяет включить наследование для динамических полей.

18.2. Управление

18.2.1. Динамические поля могут быть применены только к клиентскому объекту. Далее он может быть интерфейсно реализован как в форме/фрейме, так и в таблице.

18.2.2. Создание “Пары Динамических Полей”

18.2.2.1. Откройте меню **Dynamic fields -> Dynamic fields pair**. Создайте пару. Укажите Наименование, Класс источника (справочник Характеристик(например)), Клиентский объект цели (Клиентский объект основанный на классе, который должен быть дополнен динамическими полями (например Товары)), Таблицу со значениями (таблица, которая хранит значения Источника для Цели (Значения Характеристик для Товара).

18.2.2.2. У большинства полей есть подсказка, наведите мышкой на заголовок столбца.

18.2.2.3. Система предполагает что ключи в таблице значений соотносятся с именами таблиц источника и цели (исключая префиксы “ds_” или “d_”), например product_id и trait_id, однако их можно переопределить (**Key to target** и **Key to source**).

18.2.2.4. Система предполагает что значение будет храниться в поле с именем “**val1**”, но можно переопределить (**Value key**).

18.2.2.5. Настройки таблицы фильтров

18.2.2.5.1. Вы можете указать таблицу фильтров, тогда фильтр будет применяться на набор характеристик, при запросе данных внутри формы/фрейма (система передает parent_id). При этом, если клиентский объект грузится без **parent** (а table_for_filter не пустой), то ни одно динамическое поле не будет подгружено.

18.2.2.5.2. Укажите table_for_filter, например product_category (таблица связи, определяющая к каким категориям относится Товар).

18.2.2.5.3. Укажите parent_class_for_filter, который определит сущность, которая накладывает ограничения (Категории). Это особенно важно, если сущность иерархическая и нужно, чтобы подключались характеристики в том числе и дочерних записей относительно отфильтрованной.

18.2.2.5.4. src_key_for_filter. Определяет ключ “характеристики” в таблице фильтра, например trait_id.

18.2.2.5.5. this_and_parents_for_filter и this_and_childs_for_filter. Позволяют включить использование иерархии “вверх” и/или “вниз” по таблице фильтров. См. подсказки в интерфейсе. Например, система отфильтровала характеристики по определенной Категории, при данных настройках, система может добавить к ним Характеристики, относящиеся к родительским и/или дочерним Категориям.

18.2.2.6. Фрейм для редактора

18.2.2.6.1. Вы можете создать отдельный (клиентский объект) фрейм, который будет открываться при попытке редатировать динамическое поле. Этот фрейм будет открыт в модальном окне и ему будет передан ID записи цели (Продукта. Соответсвенно он должен быть создан на основе класса Продукт (frm_edit_product_traits)) и ID записи Источника (Характеристики). В этом фрейме вы можете просто разместить таблицу Значений_Характеристик_Для_Продукта (product_trait_value), которая будет фильтроваться по указанному Продукту и указанной Характеристике. Таким образом вам откроется таблица со значениями именно этой Характеристики для указанного Продукта и вы сможете с ней взаимодействовать, в том числе, например, менять другие поля этой таблицы, например галочку is_active (или еще что-нибудь, что будет соответствовать бизнес логике вашего проекта).

18.2.3. Настройки профайла динамических полей

18.2.3.1. Так как динамические поля добавляются “на лету”, к основной сущности, то для них нет профайла в основной сущности (Цели), а профайлы полей в таблице со значениями могут не подходить для полей подключенных динамически, хотя за основу берутся именно эти профайлы. Поэтому для них автоматически создаются свои профайлы и они располагаются в меню **Dynamic fields -> Dynamic field**.

18.2.3.2. Там вы можете переопределить Наименование поля, а также многие другие настройки для этого поля, также, как если бы вы настраивали профайл обычного поля (например сделать поле не редактируемым).

18.2.3.3. Автоматическая установка профайла исходя из данных в таблице источнике. map_source_to_dynamic_field

18.2.3.3.1. Вы можете указать объект в формате JSON, например '{"inherit":"df_is_inherit"}'. Это позволит опираясь на значение поля в таблице источника (например "inherit") проставить его как значение профайла для соответствующего динамического поля ("df_is_inherit").

19. Механизм истории

20.

20.1.

21. Фоновые задачи

21.1. apiSys

21.2. Механизм фоновых задач в самой простой своей реализации предполагает просто запуск функций по таймауту с некоторым интервалом. Эти функции могут использовать apiSYS, то есть вызов внутреннего API от имени системного пользователя, а значит вызывать любой метод любого класса. Таким образом, как правило, вся логика фоновой задаче пишется именно в методе класса, а в файле фоновой задачи он только вызывается. Например метод **Request_work**.**archiveClosed** может реализовывать архивирование заявок, которые были завершены какое-то время назад. Метод может вызываться раз в час/день/другой_период, получать все заявки которые были закрыты более чем X часов/дней, но еще не заархивированы и архивировать их.

21.3. Фоновые задачи не запускаются в отдельных процессах в рамках операционной системы. Это можно сделать при необходимости и для этого даже есть механизм (см. modules/child_processes), но по умолчанию все задачи выполняются в том же процессе.

21.4. Старт фоновых задач происходит с задержкой, а также все задачи из списка не запускаются одновременно, а каждая последующая запускается также с задержкой после предыдущей. Эти параметры можно установить в параметрах при инициализации инстанса фоновых задач, однако по умолчанию запуск идет из файла www.ts и без параметров. При этом установлены оптимальные в большинстве случаем значения по умолчанию.

21.5. Список фоновых задач определяется в modules/background_jobs/jobs.ts куда импортируются методы из файлов, которые реализуют каждую отдельную фоновую задачу из директории modules/background_jobs/jobs. Следовательно, если вы хотите разработать новую фоновую задачу, вам необходимо скопировать существующую, например modules/background_jobs/jobs/example.ts, переписать ее, написав запуск нужного вам метода и настроив интервал повторения, далее зарегистрировать эту функцию в jobs.ts

21.6. Управление из конфига

21.6.1. Вы можете управлять поведением фоновых задач из конфига (config/config.json) в разделе **bj**.

21.6.2. bj:use

21.6.2.1. Позволяет отключить запуск всех фоновых задач, если установить false.

21.6.3. bj:doNotLog

21.6.3.1. Позволяет отключить логирование всей цепочке методов вызванных из фоновых задач.

21.6.4. bj:<BackgroungJob_Name>

21.6.4.1. Позволяет точечно отключить указанную фоновую задачу.

22. Тестирование

22.1. Система на текущий момент не имеет покрытия тестами, и пока мы проводим тестирование наших модулей в процессе разработки.

22.2. В рамках разработки проектов с использованием системы вы можете использовать любой подход к тестированию, который считаете нужным и здесь мы опишем, что вам может для этого пригодится из арсенала системы.

22.2.1. API

22.2.1.1. В первую очередь вам может пригодиться API. Вы можете использовать его как по http(s), так и через подключение по сокету с использованием или без библиотеки go_core_query.См. API и go_core_query.

22.2.2. apiSys

22.2.2.1. Внутри любой функции вы можете использовать apiSys. Это функция, импортируемая из библиотеки api, которая позволяет вызвать любой метод любого класса (в терминологии ядра) от имени системного пользователя.

22.2.3. loginAsUser

22.2.3.1. Это механизм позволяющий внутри бэкенда авторизовываться от имени определенного пользователя, получать объект запроса (с его сессией) и далее выполнять любую цепочку действий.

22.2.3.2. Это позволит вам писать функции для, например, End-to-end тестирования, имитируя поведение различных пользователей.

22.2.3.3. Чтобы получить сессию пользователя, необходимо вызвать User_session.loginAsUser. В целях безопасности, вызов метода допускается только из определенных в libs/api/apiConfig методов.

22.2.3.3.1. Объект, который надо дополнять: grandAccessToLoginAsUserOperation.

22.2.3.3.2. Чтобы разрешить новый метод, из которого можно будет вызывать User_session.loginAsUser, перейдите в конфигурацию apiConfig для конкретного проекта (libs/api/apiConfig/project.ts), найдите объект grandAccessToLoginAsUserOperation_ и дополните его. Это объект, где ключ это имя класса, а значение - массив методов этого класса.

22.2.3.3.2.1. Вы можете создать, например, класс Testing и все методы (точки входа) писать в нем. Вот пример, как его методы добавить в разрешенные:

```
export const grandAccessToLoginAsUserOperation_: IGrandAccessToLoginAsUserOperation = {
   Testing: ['testOrderingRoute', 'emulateManyOrders'],
}
```

22.2.3.3.3. Также имеется объект **grandAccessToReLoginAsUserOperation**, который определяет доступ к методу **reLoginAsUser**. Этот метод, в отличии от **loginAsUser**, позволяет именно переключить текущую сессию на сессию определенного пользователя. Этот механизм может быть использован не только в тестировании, но и в основной бизнес-логике вашего проекта. Например, если часть действий пользователь может делегировать своему менеджеру, тогда такую логику можно реализовать именно через этот механизм, ограничив список, за кого можно переключится таблицей, куда, например, сам пользователь подключает “Доверенных лиц”. Такой сценарий на самом деле довольно частый и большое количество коммерческих решений его не имеют и решают эту задачу просто передачей логина и пароля другому человеку (“менеджеру”/”доверенному лицу”), что является проблемой безопасности.

23. API и go_core_query

23.1. API. Доступ по http(s)

23.1.1. Про то, как устроена авторизация и как передавать токен написано в других статьях (Механизм сессий, Документирование), но здесь повторюсь.

23.1.2. Вам потребуется выполнить запрос авторизации User.login, в ответ вы получите JWT, который требуется добавлять в Authorization header: Authorization: Bearer <Token>, к остальным запросам. Если сессия окажется просроченной или принудительно обнулена, то в ответ на любой запрос вы получите стандартный объект ответа (IAPIResponse) с кодом “-4”.

23.1.3. Запрос отправляется методом POST. Любой запрос, в том числе и на получение данных. Такой подход позволяет скрыть параметры, передаваемые в теле запроса от злоумышленников и/или провайдеров, что делает API более безопасным.

23.1.4. Любой запрос, это вызов определенного метода определенного класса. URL строится следующим образом: Адрес API, имя класса, имя метода.

23.1.4.1. Формат: /api/v1/:className/:command

23.1.4.1.1. Пример: /api/v1/user/login

23.1.4.2. Имя класса может быть с маленькой буквы.

23.1.4.3. Тело запроса это JSON объект с необходимыми параметрами. Для метода login, класса User, это login и password.

23.1.4.4. Пример curl:

```
curl -X 'POST' \
  'http://127.0.0.1:7011/api/v1/user/login' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "password": "qwerty",
  "login": "admin"
}'
```

23.1.5. Напомню что документация на каждый метод генерируется автоматически (нужно отметить метод декоратором, см. Документирование) и вы можете попробовать все методы по url **/api-docs**, а скачать openAPI файл по **/api-docs/download**

23.2. Доступ по сокету

23.2.1. Самостоятельно

23.2.1.1. Соединение по сокету использует библиотеку socket.io (на момент написания версии 4.6.1). Вы можете самостоятельно подключиться, но есть ряд нюансов. Если вам по каким то причинам потребуется использовать этот вариант, обратитесь к нам за подробностями.

23.2.2. go_core_query

23.2.2.1. Это библиотека, разработанная нами, специально для подключения к системе по сокету. Она гарантирует, что при подключении сервер получит необходимую информацию, например часовой пояс клиента, самостоятельно сохраняет токен в хранилище, умеет автоматически поддерживать авторизацию, если прописаны логин и пароль, автоматически отправлять на авторизацию, если сессия более не активна, а автоматическая авторизация отключена, перекидывать пользователя на страницу авторизации по запросу с сервера. Кроме этого библиотека следит за подключением и переподключением (используя механизмы socket.io, но со своими таймингами), обрабатывает обновление токенов, умеет сохранять uuid устройства с запросом разрешения у пользователя или без. И конечно же умеет отправлять и получать запросы к методам api, как если бы это был обычный fetch. Умеет обрабатывать некоторые виды ошибок.

23.2.2.1.1. Запросы по api (реализованные через сокет) визуально понятно логируются в консоль (если включен режим debug).

23.2.2.2. Библиотека предполагает использование в различных окружениях: браузер (import), браузер до ES6 (подключение через <script/>), nodejs, react-native (тоже через import).

23.2.2.3. В настоящее время библиотека несколько раз переписывалась под свежие версии socket.io, новые требования сервера, а также на typescript. Однако типы не были как следует описаны и могут возникать некоторые трудности при подключении различными способами. Если у вас не получается, обратитесь к нам. Так как проектов создается много, то в скором времени предполагается доработать данную библиотеку, с нормальной типизацией и отлаженными режимами. Тем не менее текущая версия работает стабильно.

23.2.2.4. Использование

23.2.2.4.1. Установите библиотеку “**npm i go_core_query**”.

23.2.2.4.2. Из библиотеки импортируется функция initGoCoreQuery, которая принимает параметры и возвращает объект { api: unknown, instance: Query }, где api это асинхронная функция запроса (принимает объект вида IAPIQuery (см. libs/api/APIStructures.ts), возвращает (через промис) ответ сервера формата IAPIResponse), а поле instance это соответственно инстанс класса, из которого вы можете получить все публичные свойства и методы, в том числе и сокет. Сам инстанс вам, как правило не понадобится, так как помимо самой функции api, обычно нужен только сокет, но чтобы подписаться используется другой механизм, а именно подписка осуществляется в функции afterInitConnect: (socket: Socket): void, которая передается через параметры.

23.2.2.4.3. Приведу листинг рабочего подключения для фронта на React, чтобы было на что ориентироваться. Если возникнут вопросы, обращайтесь к нам.

```
// @ts-ignore
import initGoCoreQuery from "go_core_query";
import { Socket } from "socket.io-client";
import { IAPIResponse } from "./structure";

export let api: (o: Record<string, any>) => Promise<IAPIResponse> = async (o) =>
{
```

```
 console.log("GoCoreContextAPIFn is not ready now. Request will be repeated after short
pause");
 await new Promise<void>(cb => setTimeout(cb, 500));

 return await api(o);
};

export let goCoreQueryInstance = {}

let logDCnt = 0;
const logDProto: (...args: any[]) => void = (...data) =>
{
 if (logDCnt >= 100)
 {
   console.log("ERR:LogDProto:No connection");
   logDCnt = 0;

   return;
 }

 logDCnt++;
 setTimeout(() =>
 {
   logD(...data);
 }, 500);
};

export let logD = logDProto;

let emitCnt = 0;
const emitProto: (key: string, ...args: any[]) => void = (key, ...data) =>
{
 if (emitCnt >= 100)
 {
   console.log("ERR:emitProto:No connection");
   emitCnt = 0;

   return;
 }

 emitCnt++;
 console.log("GoCoreAPIEmitFn is not ready now. Request will be repeated after short
pause", emitCnt);
 setTimeout(() =>
 {
   emit(key, ...data);
 }, 500);
};

export let emit = emitProto;

let socketOnCnt = 0;
const socketOnProto: (eventName: string, fn: (...args: any[]) => void) => void =
(eventName, fn) =>
{
 if (socketOnCnt >= 100)
 {
   console.log("ERR:socketOnProto:No connection");
   socketOnCnt = 0;

   return;
 }

 socketOnCnt++;
 console.log("GoCoreAPISocketOnFn is not ready now. Request will be repeated after short
pause", socketOnCnt);
 setTimeout(() =>
 {
   socketOn(eventName, fn);
 }, 500);
};

export let socketOn = socketOnProto;

let socketOffCnt = 0;
const socketOffProto: (eventName: string, fn?: (...args: any[]) => void) => void =
(eventName, fn) =>
```

```
{
 if (socketOffCnt >= 100)
 {
   console.log("ERR:socketOffProto:No connection");
   socketOffCnt = 0;

   return;
 }

 socketOffCnt++;
 console.log("GoCoreAPISocketOffFn is not ready now. Request will be repeated after short
pause", socketOffCnt);
 setTimeout(() =>
 {
   socketOff(eventName, fn);
 }, 500);
};

export let socketOff = socketOffProto;

export const initGoCoreAPI = async (params: Record<string, any> = {}) =>
{

 const {
   afterInitConnect, setIsAuth,
   socketOnConnect, socketOnDisconnect, socketOnError
 } = params;

 let debugCookie: RegExpMatchArray | string | null =
document.cookie.match(/debugGoCoreAPI=(\w+)/);
 let debug: boolean;

 if (debugCookie)
 {
   debug = debugCookie[ 1 ] === "true";
 } else
 {
   debug = false;
 }

 console.log("debugGoCoreAPI===>", debug);
 console.log("To toggle debugMode use: debug()");

 const params_ = {
   host: import.meta.env.VITE_APP_HOST,
   port: import.meta.env.VITE_APP_PORT,
   https: import.meta.env.VITE_APP_HTTPS === "true",
   path: import.meta.env.VITE_APP_PATH ?? "",
   login: import.meta.env.VITE_APP_LOGIN,
   password: import.meta.env.VITE_APP_PASSWORD,
   autoAuth: import.meta.env.VITE_APP_AUTO_AUTH === "true",

   useAJAX: false,
   tokenStorageKey: "CCSProjectNameToken",

   debug,
   debugFull: false,
   authFn: (_obj: any, cb: Function) =>
   {
     if (typeof setIsAuth === "function")
     {
       (() =>
       {
         setIsAuth(false);
       })();

       if (typeof cb === "function") cb(null);

     }
   },

   afterInitConnect: (socket: Socket) =>
   {
     if (typeof afterInitConnect === "function") afterInitConnect(socket);

     socket.on("connect", () =>
     {
```

```
      if (typeof socketOnConnect === "function") socketOnConnect(socket);

      logD = (..._data_) =>
      {
        if (logDCnt) logDCnt = 0;
      };

      emit = (key, ...data) =>
      {
        if (emitCnt) emitCnt = 0;
        socket.emit(key, ...data);
      };

      socketOn = (eventName, fn) =>
      {
        if (socketOnCnt) socketOnCnt = 0;
        socket.on(eventName, fn);
      };

      socketOff = (eventName, fn) =>
      {
        if (socketOffCnt) socketOffCnt = 0;
        logD("DEBUG: socketOff:", { eventName, fn });
        socket.off(eventName, fn);
      };

    });

    socket.on("disconnect", () =>
    {
      if (typeof socketOnDisconnect === "function") socketOnDisconnect(socket);
      logD = logDProto;
      emit = emitProto;
      socketOn = socketOnProto;
      socketOff = socketOffProto;

      socket.offAny();
    });

    socket.on("connect_error", (e: any) =>
    {
      if (typeof socketOnError === "function") socketOnError({ socket, e });
      console.log("socket.on connect_error==>");
    });

  },

};

const socketQueryOriginal = initGoCoreQuery(params_);

api = socketQueryOriginal.api;
goCoreQueryInstance = socketQueryOriginal.instance

console.log('goCoreQueryInstance===>', goCoreQueryInstance)

if (window)
{
  // @ts-ignore
  window.debug = () =>
  {
    let debug: RegExpMatchArray | string | null =
document.cookie.match(/debugGoCoreAPI=(\w+)/);

    if (debug)
    {
      debug = debug[ 1 ] === "true" ? "false" : "true";
    } else
    {
      debug = "true";
    }

    document.cookie = `debugGoCoreAPI=${ debug }`;

    console.log("debugGoCoreAPI===>", debug);
    console.log("Please, reload page to apply changes");
```

```
  };

  // @ts-ignore
  window.api = api

  // @ts-ignore
  window.goCoreQueryInstance = goCoreQueryInstance
 }
};
```

23.2.2.4.4. Ключевое здесь это initGoCoreAPI. Приведу листинг как он вызывается (там есть необязательный код, но так нагляднее), а потом прокомментирую наиболее важные параметры.

```
const init = async () => {
 initGoCoreAPI({
   setIsAuth: async (val: any) => {
     setIsAuth(val)
     setAppLoading(false)
   },
   socketOnConnect: (socket: Socket) => {
     socketRef.current = socket
     setConnected(true)
     setSocketId(socket?.id || null)
     socket.on('push', socketHandlePush)
     socket.on('command', socketHandlerCommand)
   },
   socketOnDisconnect: (socket: Socket) => {
     socketRef.current = null
     setConnected(false)
     setSocketId(null)
     socket.off('push', socketHandlePush)
     socket.off('command', socketHandlerCommand)
   },
   socketOnError: (params: { socket: Socket, e: any }) => {
     const {socket, e} = params
     console.warn('socketOnError', {socketID: socket?.id, e})
   }
 }).then()
 .catch(e => {
   console.error('ERR:initGoCoreAPI', {e})
 })
}
```

23.2.2.4.5. Рассмотрим основные параметры подключения

23.2.2.4.5.1. host, port, https, path

23.2.2.4.5.1.1. Необходимо указать хост и порт, а также используется ли https. **Важно!** Если не корректно указать useHttps, то библиотека, в текущей версии, может не выдать адекватной ошибки, но при этом будет много сообщений, которые никак не помогают решить проблему.

23.2.2.4.5.1.1.1. path как правило не используется.

23.2.2.4.5.2. autoAuth, login, password, authFn

23.2.2.4.5.2.1. Вы можете сделать так, чтобы авторизация проходила автоматически. Это используется, как правило на стороне сервера, если вы настраиваете подключения между двумя сервисами и вам выдали пользователя API для подключения. Для фронтенда обычно autoAuth=false и авторизация производится пользователем самостоятельно (см описание authFn ниже).

23.2.2.4.5.2.2. Укажите **authFn** чтобы система корректно вызывала модуль авторизации, когда получает код -4 или оповещение о выходе из системы. Это может быть например функция с переходом на страницу login. Пример:

()=>{document.location.href = "/login"}

23.2.2.4.5.3. useAJAX

23.2.2.4.5.3.1. Библиотека может работать без использования сокета, используя fetch.

23.2.2.4.5.4. tokenStorageKey

23.2.2.4.5.4.1. Под каким именем сохранять в хранилище полученный токен. Это поле можно не указывать, тогда будет значение по умолчанию. Оно актуально, если у вас несколько подключений к одному или нескольким сервисам на основе GoCore.

23.2.2.4.5.5. afterInitConnect

23.2.2.4.5.5.1. Это функция, в которой становится доступен сокет и можно повесить свои обработчики и выполнить какие-нибудь действия, которые вы хотите выполнить после подключения.

23.2.2.4.5.5.2. В первую очередь используются socket.on("connect",.. и socket.on("disconnect",.., где могут быть определены (или переопределены) функции обертки, позволяющие

работать с сокетом не передавая сокет.

23.2.2.4.5.5.2.1. Например функция emit. Если посмотреть код, то в начале она определена как функция заглушка, которая в бесконечном цикле, с таймаутом сообщает о неготовности подключения и вызывает сама себя, а внутри afterInitConnect, в подписке на подключение сокета, переопределяется и замыкает в себе этот сокет.

23.2.2.4.5.5.2.2. Также реализованы и функции socketOn, socketOff, которые позволяют в любом месте кода организовать подписку и отписку. Они также ожидают подключения сокета, пока его нет.

23.2.2.4.5.6. debug, debugFull

23.2.2.4.5.6.1. debug позволяет включить сообщения о запросах по api через библиотеку (запрос и ответ). Часто он реализован через считывание куки **debugMode** или **debugGoCoreAPI** и написана функция переключения состояния debug() (в листинге выше она тоже написана).

23.2.2.4.5.6.2. Если вам не удается подключиться, или происходит что-то нестандартное, вы можете включить debugFull, тогда библиотека будет выводить значительно больше логов, в том числе и внутренние состояния.

24. Фронт

24.1. Как работать с интерфейсами (User manual)

24.1.1. Вынести в отдельный документ

24.2. Как разрабатывать интерфейсы

25. Развертывание

25.1. Развертывание для разработки

25.2. Развертывание на сервере

26. go_core_query

26.1. Написать доку. Параметры, принцип работы, как использовать.

27. Роутинг

27.1. Основное. express, bodyParser

27.2. apiSys

27.3. мидлваре

27.3.1. cors

27.3.2. loadUser

27.4. api

27.5. needConfirm

28. Как начать

28.1. Подготовка

28.1.1. На вашем компьютере должна быть установлена MariaDB версии начиная с 10. Вы можете установить ее в основной ОС или развернуть в контейнере. Само ядро не занимается развертыванием mariadb в контейнере (есть отдельные скрипты для развертывания на серверах, но это другое). Возможно, в будущем СУБД будет развертываться автоматически.

28.1.1.1. У вас должен быть host (127.0.0.1), port (3306), login и password для доступа к СУБД. Вы можете подключаться под рутовым паролем, или создать отдельного, если на вашей машине более высокие требования к безопасности.

28.2. Получение кодовой базы

28.2.1. Если вы создаете новый проект, то вам необходимо получить копию чистого ядра и уже на его основе создавать новый проект (инициализация git, размещение в репозитории).

28.2.2. Если вы начинаете работать с существующим проектом, то вам необходимо запросить доступ к репозиторию, далее клонировать на свой компьютер.

28.3. Первичная настройка сборки

28.3.1. Выполнить первичную сборку.

28.3.1.1. Выполните запуск скрипта из package.json firstBuild, если он имеется или указанный ниже набор команд.

28.3.1.1.1. npm run firstBuild

28.3.1.1.1.1. npm i && cd public && npm i && cd .. && babel public_src --out-dir public --copy-files

28.3.1.2. В дальнейшем при разработке, вам понадобится только последняя команда, но она вынесена в отдельный скрипт npm (в package.json) - watch

28.4. Настройка конфигурации

28.4.1. config/config.json

28.4.1.1. Он исключен из git и будет создан автоматически при первом запуске. После чего его нужно дополнить и перезапустить.

28.4.1.1.1. Выполните запуск (node bin/www.js или через npm (скрипт run): “npm run run”.

28.4.1.1.2. Отредактируйте созданный конфиг.

28.4.1.2. Хост и порт запуска.

28.4.1.2.1. см. Параметры config.json -> Хост и порт запуска.

28.4.1.3. Настроить подключение к СУБД, раздел “mysqlConnection”

28.4.1.3.1. см. Параметры config.json -> Раздел “mysqlConnection”.

28.4.1.4. Остальные параметры

28.4.1.4.1. Для запуска, больше ничего менять не надо. Все параметры см. в разделе “Параметры config.json”

28.4.1.4.2. Можете запускать проект.

28.5. Запуск/перезапуск.

28.5.1. Для сборки фронтенд части ядра вы уже запускали firstBuild, но если вы будете работать с фронтенд кодом (в public_src) вам следует запустить скрипт watch: **npm run watch**. Он будет следить за изменениями и пересобирать, но он не перезагружает страницу или редактируемый блок.

28.5.2. Вы получили код с уже скомпилированным ts в js, то есть для запуска, вам не требуется запускать tsc. Однако в процессе разработки, вам следует компилировать ts в js. Для этого настройте “watcher” (в WebStorm нужно только выставить галочку “Recompile on changes” (кликните по значку TS в нижней строке состояния, он появится, если открыт ts файл)). Если не хотите использовать “watcher”, что запускайте сборку через tsc в терминале или скриптом npm run tsc.

28.5.3. Запустите основной процесс. Входной файл bin/www.js (как некогда было в Express). Запустите: “**node bin/www.js**” или “**npm run run**”.

28.5.3.1. Альтернативный конфиг

28.5.3.1.1. Вы можете запустить процесс с другим конфигурационным файлом, вместо config.json, в котором могут быть другие настройки, например указана другая база (например, копия базы с продакшена, для поиска проблемы).

28.5.3.1.2. Чтобы запустить процесс с другим конфигом, создайте его в config/ и укажите вторым параметром (команды “node”). Например: node bin/www.js prod.json

28.5.4. Перейдите по ссылке отображенной в консоле запуска. Вы должны попасть на страницу авторизации.

28.5.4.1. На чистой базе вы можете войти под администратором **admin/123**.

28.5.5. Напомню, что при первом запуске, будет создан config.json и в нем еще не будет корректного подключения к БД, и в консоле будет об этом ошибка. При повторном запуске, после настройки подключения, вероятнее всего будет заливаться БД из дампа, это занимает некоторое время, а по завершению в консоле будет сообщение, что все готово “INFO: cMysql.getConnection. Database has been filled. Enjoy your work!”. Перезапустите еще раз.

28.5.5.1. Важные сообщения при запуске, часто выведены через console.error чтобы четко выделяться, а также зафиксировать в логах (в канале ошибок) перезапуск сервера. При этом, после успешного старта, console.error возникает только при реальных проблемах. Таким образом, при запуске, несколько сообщений в console.error это нормально (но все равно прочитайте их!), а после уже не нормально и требуют особого внимания.

28.5.6. Если что-то пошло не так, внимательно читайте, что пишется в консоль, обычно становится понятно, какие проблемы возникли.

28.6. Важные замечания

28.6.1. При разработке.

28.6.1.1. Изменение бэкэнда, требует перезапуска процесса, за исключением правок в tables.json с последующей их синхронизацией через интерфейс.

28.6.1.2. Исходники фронденда располагается в public_src, а в public попадает результат сборки.

28.7. Что дальше

28.7.1. Вы можете приступать к разработке. Прочитайте раздел “Описание общей концепции” в документации.

29. Параметры config.json

29.1. Без раздела

29.1.1. Хост и порт запуска.

29.1.1.1. Параметры “host” и “port” в config.json (на первом уровне), определяют на каком IP и порту будет запущен проект. По умолчанию хост установлен в "0.0.0.0", так как это удобно для деплоймента, однако для разработки нужно сменить на "127.0.0.1" или ваш локальный IP (имеет смысл в редких случаях, когда что-нибудь отлаживаешь по сети).

29.2. Раздел “mysqlConnection”.

- host (127.0.0.1 если на той же машине, а не в контейнере)
- port - оставить как есть (3306), если только вы не запустили mariadb на другом.
- user, password. Данные пользователя в mariadb
- database. Укажите имя базы данных, для проекта.
    - База автоматически создается при первом, после успешного установления соединения с СУБД, запуске, если она не была создана ранее другими средствами. После создания, база будет залита из файла DB/ccs.init.sql или DB/<ANOTHER_INITDB_FILENAME>.sql, если ANOTHER_INITDB_FILENAME указать в поле init_sql_filename конфига (см. ниже). После заливки, еще раз перезапустите процесс.
- init_sql_filename. Определит, какой файл использовать для заливки базы при первом запуске, по умолчанию все указано корректно, но вы можете указать альтернативный файл, разместив его в директории DB/.
- dateStrings. Оставьте без изменений, эта настройка библиотеки mysql.

- utcOffset. Укажите смещение времени относительно UTC (Гринвичского времени) в минутах. Например если указано "180", то это UTC+3. Допустимы отрицательные значения для часовых поясов западнее нулевого. По умолчанию, при создании config.json ядро подставит туда значение для вашего часового пояса (часового пояса выставленного в ОС).

30. DevOps

30.1. GitFlic вместо GitHub

30.1.1. Для миграции репозитория в https://gitflic.ru/ переходим в аккаунт на GitHub (владельца репозитория), переходим в настройки аккаунта и настройки разработчика, настройки токенов (https://github.com/settings/personal-access-tokens).

30.1.2. Создаем новый токен

30.1.2.1. Выбираем нужный репозиторий и нужные права (Contents (ReadOnly)).

30.1.3. Переходим в GitFlic, создаем новый проект -> Импортировать.

30.1.3.1. Указываем логин пользователя на GitHub, полученный токен

30.1.3.2. Импортируем

30.1.4. В IDE Чтобы добавить новый remote и получить список бранчей уже с gitflic

30.1.4.1. git remote add gitflic https://gitflic.ru/repo/ВАШ_ЛОГИН/ВАШ_РЕПОЗИТОРИЙ.git

30.1.4.2. Так как в WebStorm нет интеграции с GitFlic, то мы не можем задать аккаунт в настройках. Но мы можем их задать при получении. Лучше всего использовать публичный ключ.

30.1.4.2.1. Создаем публичный ключ (в терминале (в винде я использую git bash, так как он поддерживает линуксовые команды))

30.1.4.2.1.1. ssh-keygen -t ed25519 -C "your_email@example.com"

30.1.4.2.1.2. cat ~/.ssh/id_ed25519.pub

30.1.4.2.2. Добавляем этот ключ в настройках пользователя на gitflic

30.1.4.2.2.1. https://gitflic.ru/settings/keys

30.1.4.3. Чтобы получить все бранчи пишем в терминале

30.1.4.3.1.1. git fetch gitflic

30.1.4.3.2. Далее можно работать через интерфейс, переключаясь между бранчами.

30.1.4.3.2.1. При попытке сделать checkout например ветки master из gitflic, система не дает дать ей тоже самое имя (master), для локального бранча, поэтому можно назвать gitflic-master.

30.2. Новый сервер

30.2.1. Заказать новый сервер.

30.2.1.1. Операционная система: Rocky Linux 8

30.2.1.2. Сохранить пароль

30.2.2. Подготовить код. Подготовить базу.

30.2.2.1. Определиться какую ветку будете использовать на этом сервере и убедиться что она рабочая.

30.2.2.2. Сохранить дамп соответствующей init базы (по возможности, с минимальным количеством лишних данных и без каких либо персональных, личных, нецензурных или любых других, нарушающих законодательство, данных). Лучше всего сохранить в DB/ccs.init.sql, однако можно назвать и по другому, в конфиге можно изменить.

30.2.3. Подготовить доменное имя. Прописать A запись и AAAA запись (без AAAA (IPv6) может не сработать распознавание принадлежности certbot(ом)).

30.2.3.1. IPv4 можно получить из панели или письма хостинга, а чтобы получить IPv6 подключитесь к серверу и наберите “ip addr”, найдите внешний интерфейс (там где найдете свой внешний IPv4), там же будут строки вида:
inet6 2a05:5670:135::1/32 scope global noprefixroute
Оттуда и надо взять адрес: 2a05:5670:135::1 (без /32).

30.2.3.2. Может применяться не сразу.

30.2.4.

30.2.5. Далее действуем как описано в ReadMe проекта CCS.Deployment

30.2.6. Подключиться под root и сменить пароль

30.2.6.1. Генерируем новый пароль, например набрав: pwgen -1 32 -cny

30.2.6.1.1. Сохраняем его себе

30.2.6.2. Меняем пароль: passwd

30.2.7. Снова подключаемся под root (уже для установки)

30.2.7.1. Установим вручную vim: dnf install -y vim

30.2.7.2. Создать prepare.sh и скопировать в него содержимое файла prepareOS/prepare.sh

30.2.7.2.1. vim prepare.sh && chmod u+x prepare.sh

30.2.7.2.2. ./prepare.sh

30.2.7.3. Скрипт установит необходимый софт и предложит создать пользователя

30.2.7.3.1. Создайте пользователя, под которым будут работать сервисы.

30.2.7.3.1.1. Например ccs_app. Лучше называть по имени проекта, чтобы когда у тебя много серверов и много проектов сразу понимать где ты и с чем работаешь.

30.2.7.3.2. Сгенерить пароль: pwgen -1 32 -cny

30.2.7.3.2.1. Сохраните пароль!

30.2.8. Настройка конфига

30.2.8.1. На основе конфига будут установлены и запущены сервисы в отдельных **podman** контейнерах.

30.2.8.2. Вы можете прописать все сервисы которые должны быть развернуты. Каждый сервис разворачивается опираясь на определенный шаблон. Есть свой для базы, для бэкапов, для nginx, для coturn, а также для приложений на GoCore.

30.2.8.2.1. Если потребуется еще какой-то софт, для них также можно будет написать шаблоны.

30.2.8.3. Пример конфига лежит в проекте CCS.Deployment/config_template.sh и именно он, если не создать отдельный (см. подпункт ниже), будет скопирован в config.sh при первом запуске скрипта start.sh

30.2.8.3.1. Для удобства, вы можете в проекте CCS.Deployment (можно в отдельном бранче), в разделе services создать новую директорию по имени сервиса, который вы укажете в конфиге (ccs_<project_name>). В нее вы можете скопировать CCS.Deployment/config_template.sh и настроить его под этот проект.

30.2.8.3.1.1. Вы также можете создать там config_template.json, тогда именно он будет взят за основу для вашего app. Только оставьте в нем переменные вида REPLACE_<имя_параметра>, чтобы при установке подставились все значения, относящиеся к этому сервису. Если не создать такой файл, то будет взят из app_template/config_templare.json

30.2.8.4. Параметры для каждого сервиса в конфиге начинаются с имени сервиса, для которого этот параметр указывается.

30.2.8.5. При запуске установки, создается или дополняется файл .env, и именно в нем вы сможете найти рутовый и пользовательский пароль к сервису mariadb (если вдруг потребуется).

30.2.8.5.1. Важно! Если вы не с первого раза прописали в конфиге все правильно и захотите изменить параметры, то изменения некоторых из них (затрудняюсь сказать каких именно) может не привести к изменениям. В этом случае, удалите соответствующие записи из .env (скопируйте его перед этим на всякий случай).

30.2.8.6. Перечислите нужные вам сервисы в services:

```
declare services=("db" "ccs_app" "ccs_app_front" "nginx" "backup")
```

30.2.8.7. В примере указаны сразу два сервиса на nodejs, для бэкэнда и для фронта (бэкэнд использует шаблон “app_template”, а фронт “static_template” - как они работают можно посмотреть в CCS.Deployment).

30.2.8.7.1. Если вам не нужен фронт, удалите его из services и все его параметры (ccs_app_front_<param_name>=””)

30.2.8.8. Настройте параметры

30.2.8.8.1. URL гита с доступом по ssh (начинается на git@, а не на https://). ccs_app_git_url

30.2.8.8.2. Укажите бранч. ccs_app_git_branch

30.2.8.8.3. Укажите имя дампа (должен лежать в проекте в DB/). ccs_app_init_sql="ccs.init.sql"

30.2.8.8.4. Укажите домен. ccs_app_domains

30.2.8.8.4.1. Или домены через пробел ccs_app_domains=(“domain1” “domain2”)

30.2.8.8.5. В параметре ccs_app_nginx_location можно указать какие маршруты nginx не должен обрабатывать как статику, а проксировать в приложение.

30.2.8.8.5.1. По умолчанию там уже все прописано, но можно дополнить.

30.2.8.8.6. Если у вас есть сервер бэкапов, то пропишите его

```
ccs_app_backup_files_remote_address=("myuser@192.168.1.100")
```

30.2.8.8.6.1. В процессе установки система попросит добавить на том сервере публичный ключ этого сервера, чтобы проходила авторизация.

30.2.8.8.7. Остальное пока не описано. Если не ясно из названия и примера, напишите нам.

30.2.9. Теперь подключимся под созданным пользователем.

30.2.9.1. Важно! Именно открыть новый терминал, а не переподключиться через su

30.2.9.2. Создать и получить ключ: ssh-keygen && cat ~/.ssh/id_rsa.pub

30.2.9.3. Добавить ключ в репозиторий этого проекта (CCS.Deployment)

30.2.9.4. Получить проект

30.2.9.4.1. cd ~/ && git clone -b master git@github.com:CCSMSKRU/CCS.Deployment.git

30.2.9.4.2. Если вы создавали отдельный бранч для конфигурации своего проекта, то вместо master подставьте свой.

30.2.9.5. Копируем start.sh в корень (из шаблона), чтобы вынести из под гита и выдаем доступ на исполнение

30.2.9.5.1. cp ~/CCS.Deployment/start_template.sh ~/start.sh && chmod u+x ~/start.sh

30.2.9.6. *Если нужно обновить файлы deployment, то делаем git pull и снова копируем

30.2.9.6.1. cd ~/CCS.Deployment/ && git pull && cd && cp ~/CCS.Deployment/start_template.sh ~/start.sh && chmod u+x ~/start.sh

30.2.9.7. Запускаем (./start.sh), настраиваем конфиг (см. ниже), выставляем config_is_ready в 1, запускаем еще раз

30.2.9.7.1. ./start.sh install db ccs_<project_name> nginx backup

30.2.10. Установка и запуск сервисов.

30.2.10.1. После настройки конфига, выставьте в нем config_is_ready=1, сохраните и запустите установку:

```
./start.sh install db ccs_app ccs_front nginx backup
```

30.2.10.1.1. Перечислите сервисы которые хотите установить.

30.2.10.1.2. В ходе установки, следуйте указаниям скрипта.

30.2.10.1.3. В ходе установки будет залита база.

30.2.10.1.4. В ходе установки nginx сервис будет запущен на http, а не на https. Чтобы подключился https нужно будет выпустить сертификаты (об этом ниже) и после запустить установку nginx заново.

30.2.10.1.5. После установки, вы можете открыть и отредактировать config.json самого проекта (не забудьте перезапустить сервис после этого вместе с nginx (./start.sh update <имя_сервиса> && ./start.sh restart nginx))

30.2.10.1.5.1. Он находится в ~/apps/<имя_сервиса>/config

30.2.10.1.5.2. Можно настроить в WebStorm Remote Host или использовать vim/nano.

30.2.11. Установка сертификатов

30.2.11.1. Заходим под root.

30.2.11.1.1. Создаем и заполняем cert.sh. Его можно скопировать из CCS.Deployment и выдать права, но легче через vim просто вставить его содержимое:

30.2.11.1.1.1. vim ./cert.sh && chmod u+x cert.sh

30.2.11.1.2. Запускаем его указав имя пользователя под которым хранится config.sh

30.2.11.1.2.1. ./cert.sh ccs_app

30.2.11.1.3. Следуем указаниям скрипта.

30.2.11.1.4. Снова заходим под основным пользователем и переустанавливаем nginx

30.2.11.1.4.1. ./start.sh install nginx

30.2.12. Дополнения

30.2.12.1. При перезапуске приложения также требуется перезапустить nginx. ./start.sh restart ccs_app nginx

30.2.12.2. Другие опции для start.sh:

30.2.12.2.1. Просмотр логов

30.2.12.2.1.1. ./start.sh log <имя_сервиса>

30.2.12.2.2. Перезапуск сервиса

30.2.12.2.2.1. ./start.sh restart <имя_сервиса> nginx

30.2.12.2.3. Обновление

30.2.12.2.3.1. ./start.sh update <имя_сервиса>

30.2.12.2.4. Обновление и сборка

30.2.12.2.4.1. ./start.sh update_and_npmbuild <имя_сервиса>

30.2.12.2.5. Обновление, установка зависимостей и сборка

30.2.12.2.5.1. ./start.sh update_and_npmi_and_npmbuild <имя_сервиса>

30.2.12.2.6. Не забывайте перезапускать nginx после обновления бэкэнда

30.2.12.2.6.1. ./start.sh restart nginx

30.2.12.2.6.2. А лучше сразу обновлять и перезапускать одной строкой

./start.sh update <имя_сервиса> && ./start.sh restart nginx