elu/dia

Предисловие

Настоящий документ представляет собой сборник документации по нескольким взаимосвязанным, но относительно самостоятельным проектам:

- общей спецификации https://github.com/do-/elu_dia_docs;
- клиентской библиотеки https://github.com/do-/dia.js;
  с адаптерами к
  - w2ui https://github.com/do-/elu_w2ui;
  - и SlickGrid / jQueryUI https://github.com/do-/elu_w2ui
- и серверной библиотеки https://github.com/do-/dia.js;

Всё это в комплексе представляет собой открытую платформу для разработки корпоративных информационных систем с Web-интерфейсами. Открытую -- во многих смыслах. В частности, это значит, что ни одна из перечисленных реализованных программных компонент не является обязательной и незаменимой.

Описанная здесь серверная библиотека Dia.js, функционирующая в экосистеме node.js, изначально писалась на Perl5, у неё есть аналог на java, завтра она может оказаться портированной на Python или ещё что-то. Спецификация достаточно проста, чтобы писать Web-сервисы в соответствии с ней и без какого-либо специального ПО.

Клиентская библиотека elu.js, со своим понятием о компонентах и своим механизмом HTML-шаблонов, как может показаться, во многом дублирует функциональность гораздо более известных "framework"'ов. Если кто-либо сочтёт предпочтительным AngularJS, ReactJS или что-то похожее -- ничто не может помешать ему использовать из elu.js одну-две функции для работы с AJAX-запросами или обойтись вовсе без неё -- и полноценно работать с сервером на Dia.js или её аналоге.

Несмотря на описанную гибкость, есть одна вещь, которая объединяет elu и все варианты dia: это описанная в спецификации система координат, в которой предлагается рассматривать все решаемые задачи. Первичен именно этот набор базовых переменных, он определяет и формат параметров HTTP-запосов, и правила именования файлов, и многое другое в описанных библиотеках. С их помощью легко следовать определённой дисциплине, а она, в свою очередь, помогает экономить время при разработке и поддержке.

Как отмечено выше, здесь под одной обложкой собрана документация из нескольких испточников. Каждый из них функционирует как самостоятельный wiki-ресурс, а общий документ собирается автоматом. При таком подходе неизбежны проблемы в виде нестыковок, повторений, провалов. Тем не менее, автор надеется, что полученный таким образом текст всё же может оказаться полезным для последовательного ознакомления с материалом и просит читателей сообщать об обнаруженных неточностях, используя предназначенные для этого механизмы указанных общедоступных репозиториев.

Устройство Web приложения

Ниже изложены общие требования к разрабатываемым информационным системам. Они не привязаны ни к специфической предметной области, ни к конкретным программным платформам (операционным системам, базам данных и т. п.).

Базовые технические положения сводятся к использованию:

- HTML для построения интерфейса пользователя;
- HTTP для передачи данных

с учётом того обстоятельства, что любой визуальный эффект, достижимый средствами HTML, доступен путём исполнения js-кода, а обработка отображаемых данных на клиенте обходится дешевле, чем исполнение такого же алгоритма на сервере.

Общие положения

Приступая к проектированию ИС, следует сразу принять во внимание тот факт, что URI, будучи элементом транспортного протокола, виден клиенту-человеку как минимум в адресной строке браузера. С немалой вероятностью — и во внешних документах (например, извещениях, рассылаемых по e-mail, SMS и т. п. самой системой). Не говоря уже о рекламных материалах типа футболок, ручек, пакетов и т. п. И то, как этот адрес выглядит, влияет на оценку работы со стороны заказчика.

В то же время можно с уверенностью сказать, что вид URL важен лишь для GET-запросов, в ответ на которые клиент ожидает получить Web-страницы либо бинарные (PDF и т. п.) файлы с фиксированным содержимым. Что же касается AJAX-трафика — его можно считать скрытым от глаз благонамеренного пользователя (вопросы безопасности здесь не рассматриваются).

Соответственно, необходимо с самого начала разделить все URI, используемые в системе, на 2 категории:

- "красивые": пригодные для показа клиенту — для них следует предусмотреть высокоуровневую обработку;
- "технические": пригодные для передачи произвольных данных — для них следует сразу выделить корневой путь, заведомо неприемлемый для "красивых" URI (например: /_/).

Из соображений безопасности оба класса должны быть явно описаны в конфигурации. Для всех прочих запросов надо выдавать ошибки с кодами 4xx.

Статические и динамические запросы

Все HTTP-запросы, генерируемые клиентами, разделим на две категории: статические и динамические.

Статическим называется GET-запрос, ответ на который в рамках каждой фиксированной версии ИС определяется исключительно path-частью соответствующего URI.

То есть для любого клиента в любое время, пока на сервере опубликована фиксированная версия ИС, на статический запрос с одним path выдаётся идентичное содержимое. В подавляющем большинстве случаев это содержимое файла, путь которого в файловой системе, видимой Web-серверу, соответствует path-части URI с точностью до замены префиксов ("корней"). Однако в настоящем документе "статический" не обязательно означает "лежащий на диске": некоторые из таких запросов могут выдаваться с использованием "тяжёлого" сервера и из БД, и из иных источников.

В двух предыдущих абзацах "ответ" и "содержимое" считаются идентичными вне зависимости от алгоритма сжатия, который может определяться переданным заголовком Accept-Encoding.

Любой запрос, не являющийся статическим — мы будем называть динамическим. URI любого динамического запроса относится к категории "технических". Обратное — неверно.

"Лёгкий" и "тяжёлый" сервер

Северная часть ИС должна содержать минимум 2 сервера:

- "лёгкий":
  - принимает HTTP-запросы;
  - может трансформировать некоторые их компоненты в соответствии с регулярными выражениями, описанными в конфигурации;
  - либо выдаёт HTTP-ответ из файловой системы;
  - либо передаёт запрос другому серверу по HTTP или иному протоколу (FCGI и т. п.)
- "тяжёлый":
  - принимает запросы от "лёгкого" сервера;
  - формирует HTTP-ответ, исполняя сколь угодно сложный алгоритм с использованием произвольных дополнительных источников данных.

"Тяжёлый" сервер обрабатывает все динамические и некоторые статические запросы; "лёгкий" — только статические.

В качестве "лёгкого" сервера оптимально использовать специально разработанные для этой цели nginx или lighttpd; иногда (по требованиям совместимости) приходится иметь дело с Apache httpd или ещё чем-либо.

Способ обработки динамических запросов здесь не рассматривается, так что никаких технических деталей о "тяжёлых" серверах приведено не будет.

Конфигурация "лёгкого" сервера как часть кода ИС

Файлы, называемые "конфигурационными" для nginx, lighttpd, httpd и прочих HTTP-серверов вовсе не сводятся к наборам параметров. По сути они являются программным кодом на специфических скриптовых языках. Там описываются алгоритмы преобразования запросов, есть возможность определять и использовать переменные, имеются условные операторы, поддерживается модульность.

Здесь важно отделять логику, специфичную для приложения (правила коррекции URI, внутренние перенаправления, проксирование, приписывание заголовков и т. п.), от параметров конкретной установки ИС (имена серверов, порты, абсолютные пути и т. п.)

Соответственно, программный код ИС должен включать по отдельности:

- рабочий локальный файл (app.conf), предназначенный для включения (include) в общую конфигурацию;
- файл-пример (virtual-host.conf.orig) с комментированным фрагментом общей конфигурации, предназначенный для копирования и правки руками ситемного администратора.

В основном администратор должен копировать содежимое virtual-host.conf.orig в общую конфигурацию и править его по месту только при первичной установке ИС на новой машине. Хотя возможны дальнейшие правки по ходу изменения конфигурации сети. Разработчик должен понимать, что правка virtual-host.conf.orig как таковая автоматически не приведёт ни к каким изменениям ни на одном сервере, который он не администрирует лично. Переконфигурация возможна только через административные процедуры.

А app.conf может меняться в каждой новой версии ИС и копируется при её установке на каждый сервер. Он должен быть одинаково применимым для всех экземпляров данной ИС и, соответственно, не может содержать привязки ни к одному параметру (абсолютная директория, порт и т. п.) отдельно взятой машины.

Разумно предусмотреть несколько версий app.conf для разных вариантов работы приложения. В частности:

- для штатного режима (собственно app.conf);
- для режима поддержки/обновления (app-maintenance.conf), в котором на любой динамический запрос должен выдаваться ответ со статусом 503 Service Unavailable, чтобы js-обработчик отобразил страницу о временной недоступности системы.

Статические запросы

Общие требования

Статические запросы должны отправляться на сервер с глаголами HEAD или GET. На прочие глаголы следует выдавать ошибку 405 Method not allowed.

Для подавляющего большинства статических запросов содержимое ответов может быть размещено в файловой системе, являясь частью программного кода ИС (с точки зрения версионного репозитория). Такие ответы могут и должны выдаваться "лёгким" сервером самостоятельно.

Кэширование

Любой браузер запоминает (кэширует) полученные ответы на GET-запросы и старается по мере возможности переиспользовать их без повторных обращений к серверу. HTTP предолагает широкий набор заголовков для гибкого управления кэшированием на клиенте, однако практика показывает, что для URI с непустой частью "имя файла.расширение" полагаться на них нельзя: любой браузер может использовать свою эвристику поверх стандарта.

Поэтому все статические файловые ответы должны быть чётко разделены на 2 группы:

- не кэшируемые (сразу просроченные) — для них следует выдавать заголовок Expires в прошлом;
- кэшируемые навечно — их надо сопровождать Expires в далёком будущем, но при формировани URI всегда приписывать параметр, гарантированно меняющийся с обновлением версии ИС (либо чаще — например, при каждом входе пользователя в систему).

Фиксированые (hardcoded) URI

Любая ИС должна поддерживать адекватное обслуживание URI, жёстко прописанных во внешних HTTP-клиентах. Это:

- /favicon.ico — логитип системы
- /robots.txt (http://www.robotstxt.org/orig.html)

Кэширование обоих следует исключить.

Мини-сайт для неподдерживаемых браузеров

Для рассматриваемых здесь ИС необходимо использование браузеров с поддержкой определённых js API. В то же время невозможно предотвратить обращения к Web-серверу роботом, текстовым браузером (типа lynx), устаревшим браузером, либо браузером, у которого исполнение js-кода заблокировано: глобально или по месту; стандартными опциями либо сторонними средствами. Раз это невозможно предотвратить — необходимо предусмотреть.

Поэтому ИС должна содержать Web-страницу с сообщением о несоответствии используемого браузера требованиям ИС и рекомендациями о том, как решить данную проблему.

Она должна быть свёрстана в дизайне ИС, но без js, css, средствами только HTML 3.2 или ниже, с GIF вместо PNG — так что представляет собой изолированный мини-сайт.

Эту страницу следует кэшировать. При редактировании (например, редизайне) несложно изменить URL.

Если предусмотрено несколько разных сообщений (об отключённом js — одно, о несоответствии версии браузера — другое), то для каждого необходимо сверстать отдельную страницу.

Ресурсы HTML

Здесь упомянуты наиболее известные типы ресурсов, используемых HTML и выносимых в отдельные файлы: css, js, графика.

Все они должны кэшироваться навечно. Соответственно, в URI каждого из них (например, в имя корневой директории) должен быть заложен фрагмент, предотвращающий кэширование между версиями системы.

css

Все презентационные свойства элементов, отображаемых на экранах ИС, которые в принципе могут быть вынесены в CSS — должны быть определены именно там. В частности, доступные элементы векторной графики (границы со скруглениями углов, тени, градиентная заливка и т. п.) должны определяться в CSS, а не верстаться при помощи пиксельных изображений.

Запрещается использование атрибутов STYLE (в том числе через js), все свойства должны определяться только через CLASS.

Крайне желательно использование LESS (http://lesscss.org/) и прочих аналогичных средств, минимизирующих потребность в дублировании внутри CSS-кода, не порождая большой зависимости от дополнительного ПО (как SASS http://sass-lang.com/, в оригинале привязанный к Ruby).

js

В клиентском js-коде ИС должен быть реализован максимум всей прикладной функциональности.

Если некая логическая функция может быть вычислена на клиенте (например, при проверке введённых данных) — не следует по этому поводу обращаться на сервер.

Если некий небольшой объём данных (JSON до 5 Мб: набор пунктов меню и т. п.) используется много раз за сессию и не может меняться без запроса со стороны пользователя — значит, следует хранить его в Web Storage, а не перезапрашивать каждый раз.

Изображения и multimedia

Всё графическое, звуковое и видеосодержимое, доступное клиенту ИС, делится на 2 сорта:

- дизайн (часть кода ИС, общего для всех инсталляций);
- хранимые данные (свои на каждой среде).

графические элементы дизайна

Из элементов общего дизайна страниц ИС:

- крупные (> 100 x 100px) полноцветные (trueColor) изображения изображения (фоны, "плашки" и т. п.), должны храниться в формате JPEG;
- прочие (мелкие фрагменты с ограниченной палитрой) — в формате PNG.

Наборы однотипных пиктограмм (кнопок, статусов и т. п.), которые с большой вероятностью требуются клиенту все вместе, желательно компановать в общие файлы (технлогия "CSS image sprite", https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Images/Implementing_image_sprites_in_CSS).

Всё перечисленное в данном пункте, а также специфические типы графики (шрифты, курсоры) требует вечного кэширования.

нетекстовые данные системы

ИС может предполагать выдачу общедоступных изображений, входящих не в код системы, а в данные (иллюстрации к статьям, логотипы организаций и т. п). Такие запросы являются статическими, хотя могут выдаваться и "тяжёлым" сервером из БД или иного источника, в любом случае для них требуется обеспечивать кэширование. Если содержимое размещено в файловой системе, необходимо, чтобы оно не попадало в область видимости системы контроля версий ИС.

Хранение фото сотрудников, скан-копий личных документов и т. п. конфиденциального содержимого рядом с компонентами дизайна ИС недопустимо. Такого рода файлы должны выдаваться только динамическими запросами, с проверкой прав пользователя.

HTML-файлы

Традиционно HTML-файлы рассматриваются в основном как страницы с неизменяемым содержимым, загружаемые в браузер целиком и непосредственно: как вышеописанная страница для неподдерживаемых браузеров.

На современном уровне развития технологий представляется целесообразным, чтобы:

- страницы, загружаемые в браузер непосредственно (по URI в строке адреса), были привязаны к данным (через динамические AJAX-запросы);
- неизменяемый HTML-код непосредственно загружался не в окно браузера, а в невидимый буфер посредством статического AJAX-запроса и далее отображался клиенту после обработки js.

HTML-фрагменты

"Фрагментами" здесь называются HTML-страницы, не предназначенные для загрузки с отображением в каком-либо окне браузера (главном либо фрейме).

js-код получает такие файлы в ответах на AJAX-запросы, обрабатывает и потом отображает встраивает в ту или иную область страницы.

Например, login-форма, которую пользователь может увидеть, изначально обращаясь по произвольному URI (например, по ссылке из письма, с закладки и т. п.), не должна иметь собственного адреса, на который клиент бы перенаправлялся. Желая открыть карточку задачи и сделав для этого запрос на /tasks/12345/, клиент может увидеть форму ввода логина и пароля, но в адресной строке будет /tasks/12345/, а HTML-код формы будет загружен js-кодом, который на старте (onLoad) обнаружит отсутствие актуальной сессии.

Когда пользователь введёт login/пароль, js-код прежде всего визуально обозначит начало процесса (затемнение, вращающийся индикатор, и т. п.) и js-код попытается завести сессию при помощи динамического AJAX-запроса. Если ответ будет успешным, то дальше будет инициировано ещё 2 AJAX-запроса:

- динамический — на данные задачи;
- статический — на HTML-фрагмент с пустой карточкой.

Получив оба ответа, js-код подправит HTML и отобразит карточку уже с данными.

На второй и последующие запросы на карточки задач браузер будет уже не загружать HTML-фрагмент, а доставать его из кэша (если, конечно, клиен не отключит или не опустошит его специально).

Главная страница

HTML-код, выдаваемый в ответ на запросы с "красивыми" URI, должны быть весьма компактным и универсальным. Он должен сводиться только к:

- перенаправлению (META REFRESH) на аварийную страницу в случае недоступности js (NOSCRIPT);
- установке корневого URI для относительных ссылок (BASE HREF);
- загрузке CSS-файлов и js-библиотек.

Остаток настоящего раздела в касается того, какие действия должна выполнять js-функция, запускаемая по событию onLoad. Без ограничения общности мы предполагаем, что эта функция — общая для всех страниц ИС с "красивыми" URI. Да и HTML-файл (назовём его handler.html) на всю систему нужен всего один: он назначается на все запросы в конфигурации "лёгкого сервера", а обработка конкретного URI начинается уже в js-коде.

Итак, js по событию onLoad должен:

- Прежде всего, завершить проверку совместимости браузера и доступности всех небходимых функций. При обнаружении конфликта — перенаправить клиента на аварийный мини-сайт. Если же проверка пройдена — запомнить этот факт в Web Storage (для надёжности — sessionStorage), чтобы в следующий раз не тратить на это времени.
- По содержимому адресной строки браузера (то есть URI) и состоянию Web Storage определить объект $_REQUEST с основными параметрами запроса (подробнее см. раздел о динамических запросах): типом экрана (type) и номером карточки (id), а также объект $_USER с реквизитами текущего пользователя, которые могут использоваться на клиенте — её содержимое оптимально хранить в Session Storage, изначально получив в успешном ответе при входе в систему.
- Проверить полученные параметры на допустимость. В частности, если ИС требует обязательной аутентификации, а сессия не заведена — выбросить соответствующее исключение.
- Найти среди имеющихся js-функций подходящую для текущего типа экрана/номера объекта. Если таковой не обнаружится — выбросить соответствующее исключение.

- Если алгоритм дошёл до данного пункта — запустить найденную функцию-обработчик.
- При перехвате исключения — запустить его обработку. В частности:
  - если по URI был определён неизвестний тип или недопустимый номер — загрузить и отобразить HTML-фрагмент, соответствующий HTTP-ошибке "404 Not Found".
  - если пользователь не представлен системе, а требуется сессия — загрузить login-форму (пример см. в разделе об HTML-фрагменте) или инициировать иной сценарий входа в систему.

Далее работа WebUI полностью определяется упомянутыми функциями-обработчиками. Их взаимодействие с сервером осуществляется посредством динамических AJAX-запросов (см. ниже). Есть общие требования к их оформлению:

- Для каждого AJAX-запроса должно быть установлено конечное время ожидания.
- Обработку перенаправлений (3xx) следует запретить: внутри ИС, тем более для AJAX-запросов, такие ответы выдаваться не должны. Получение статуса 3xx следует рассматривать как ошибку.
- Перед запуском AJAX-запроса js должен визуально обозначить начало процесса путём затемнения/снижения контраста всего экрана или отдельной области, показа индикатора прогресса, замены указателя "мыши" и т. п. Необходимо сразу предусмотреть, чтобы этот визуальный эффект гарантированно снимался бы при любом, в том числе аварийном, результате вызова.
- По окончании вызова прежде всего необходимо определять аварийные ситуации, когда ответа нет в принципе (сеть недоступна, истекло время ожидания и т. п.) и предпринимать соответствующие действия, в норме:
  - для запросов на показ данных — повторять запрос (со счётчиками-ограничителями);
  - для запросов на активные операции — отображать ошибку.
- Далее, если получен HTTP-ответ — проверить его статус и, если он отличен от 200 OK, обработать данное исключение. Например:
  - 503 Service Unavailable, то есть на сервере проводятся работы по обновлению — отобразить сответствующее сообщение. В идеале такой ответ должен содержать JSON-тело со значением плановой даты/времени окончания работ, а сообщение — таймер обратного отсчёта до перезагрузки страницы.
  - 401 Unauthorized, то есть клиент передал номер сессии, которой (уже) нет на сервере — стереть информацию о пользователе из Session Storage и инициировать обновление текущей страницы, что должно привести к запуску сценария аутентификации.
  - Для прочих кодов, кроме 200 OK (использование 10[01] и 20[1-6] не рекомендуется в принципе) — отображать ошибку.
- Затем проверить заголовки ответа (Conent-Type = 'application/json', Conent-Size положительный) и при обнаружении отклонений — отображать ошибку.
- Попробовать расшифровать тело ответа как JSON, при невозможности — отображать ошибку.
- Если алгоритм дошёл до этого пункта — продолжать предусмотренную обработку полученных данных.

Требования к системе могут предполагать, что, пока открыт браузер, пользовательская сессия не должна заканчиваться, даже если клиент не генерирует никаких событий ввода. Для этого при загрузке страницы нужно запускать js-таймер с периодом менее времени жизни сессии (14 минут для 15-минутного предела), который инициирует специальные пустые динамические запросы. Чтобы не возникало расхождений с настройками сервера, js-код должен получать сконфигурированное там значение времени жизни сессии из AJAX-ответа при успешной аутентификации. Для снижения нагрузки на сеть и сервер, перед запуском каждого AJAX-запроса этот таймер необходимо отменять и запускать заново — тогда лишних запросов передаваться не будет.

По соображениям безопасности, возможны и противоположные требования (чтобы сессия не продлевалась автоматически) — тогда аналогичный таймер тоже нужен, но он должен не слать AJAX-запрос, а показывать предупреждение с обратным отсчётом времени и кнопкой, по которой пользователь может продлить сессию явно.

Динамические запросы

Сессии пользователей

Если ответы ИС зависят от личности человека, запустившего браузер, значит, в каждом запросе должна быть информация об этом пользователе.

Как правило, используется следующий компромисс между скоростью и безопасностью: в начале работы клиент представляется системе (вводит login, пароль и/или прочие данные аутентификации), в успешном ответе на этот запрос браузер получает Cookie (https://www.w3.org/Protocols/rfc2109/rfc2109) с длинной случайной строкой, которая далее в течение непрерывного сеанса работы используется как идентификатор пользователя, не требующий подтверждения.

Для поддержки сессий (генерация номера, хранение связи с users.id, контроль времени жизни) ИС может либо использовать готовый API, предоставляемый платформой (Java Servlets, ASP, PHP и т. д.), либо реализовать свой собственный. В последнем случае необходимо иметь в виду https://www.owasp.org/index.php/Session_Management_Cheat_Sheet.

В частности, значение номера сессии должно оставаться неизвестным для клиентского js-кода. Информацию о личности пользователя он должен получать в ответе на AJAX-запрос (оптимально — при аутентификации).

Общий формат запросов

Динамические запросы должны отправляться на сервер с глаголом POST. На иные глаголы следует выдавать ошибку 405 Method not allowed.

За исключением особых случаев, URI для всех динамических запросов должен быть один (например, /_/back/).

Основные (см. следующий раздел) параметры запросов должны передаваться в адресной строке: /_/back/?type=users&id=1&action=update

Желательно конфигурировать логирование таким образом, чтобы рядом с URI записывались и сессионные Cookies, и тела POST-запросов.

Соответственно, конфиденциальные данные, которые необходимо передавать с клиента на сервер, но нельзя записывать в log-файлах (например, введённое значение пароля) должны пересылаться не в URI и не в теле, а в дополнительных заголовках HTTP-запросов.

Остальная информация, передаваемая с клиента на сервер, должна быть представлена в формате JSON в теле POST-запроса. Рекомендуется, чтобы это в любом случае был JSON-объект, элементами которого являются объекты, сответствующие множествам элементов ввода (формам) на экране. В частности (с точностью до форматирования):

- тело запроса на обновление наименования элемента справочника:

```
{
   data: {
       "label": "New Label"
   }
}
```

- тело запроса на поиск первых 15 записей по строке "абв":

```
{
   search: {
       "q": "абв",
       "pager": {
           "start": 0,
           "portion": 15
       }
   }
}
```

Текст JSON должен передаваться в минимальном формате, без незначащих пробельных символов.

Основные параметры

Тип (type)

Это символическое имя **типа** данных, поддерживаемого системой: например, 'users' для пользователей, 'tasks' для поручений и т. п. Если данные системы хранятся в основном в одной БД и некоторый экран предназначен для просмотра/редактирования записей из определённой таблицы (скажем, news для новостей) — оптимально использовать её имя и при формировании "красивого" URI (/users/) и для динамических запросов (/_/back/?type=news&t=179833205&_=0.38723875246).

Если параметр type не указан — по данному запросу только продлевается время жизни текущей сессии и выдаётся успешный ответ, либо, если сессия не определена, возвращается 401 Unauthorized.

Номер объекта (id)

Это видимый клиенту-человеку уникальный **номер** объекта данных ИС, передаваемый в динамический запрос без изменения.

В простейшем случае он может совпадать со значением поля-счётчика, используемого как первичный ключ в таблице БД

```
/documents/673/
/_/back/?type=documents&id=673&t=179833205&_=0.38723875246
```

но при этом легко могут возникнуть проблемы безопасности по поводу перебора значений (URL hacking).

Использование GUID или иного случайного синтетического идентификатора:

```
/accounts/df51cdd9-b980-4c23-b034-3d16b36676ac/
/_/back/?type=accounts&id=df51cdd9-b980-4c23-b034-3d16b36676ac&t=179833378&_=0.385165202656)
```

делает подбор адреса бессмысленным и выглядит уже довольно серьёзно, но не идеально для стороннего использования URI системы.

Для публичных ИС, чьи внутрение адреса могут встречаться в сторонних текстах, оптимально генерировать, запоминать и использовать специальные человекочитаемые идентификаторы, включающие естественное наименование объекта:

```
/products/vaz-2106/
/_/back/?type=products&id=vaz-2106&t=179843579&_=0.1)
```

но это, по понятным причинам, повышает трудоёмкость разработки.

Какой id использовать в данной ИС для каждого типа — необходимо выбирать в каждом конкретном случае. Так или иначе, пара type/id должна идентифицировать объект системы однозначно.

Параметр id может быть не указан, в этом случае запрос касается типа в целом: обычно это либо поиск для отображения списка, либо действие (см. ниже) создания нового объекта.

Действие (action)

Символическое имя **действия**, которое требуется произвести с хранимой на сервере информацией по ходу данного запроса.

Этот параметр пуст для запросов на извлечение данных (как правило, с id — данные карточки документа, без id — список), не требующих открытия транзакций, и непуст для активных операций, таких как создание (create), редактирование (update), удаление (delete) и т. п., где приходится обеспечивать целостность данных.

Часть (part)

Символическое имя **части** содержимого, которую требуется извлечь в результате данного запроса.

Этот параметр несовместим с action и, соответственно, может быть непустым только для запросов на извлечение данных.

Он применяется в тех случаях, когда объект, заданный парой type/id, содержит объёмные данные, необходимые для отображения на одних экранах и ненужные на других. Кроме того при одновременном использовании различных js-библиотек, может потребоваться представлять разные части содержимого одного объекта в разных JSON-структурах (имена полей, вложенность объектов).

Пример:

- /_back/?type=users&id=12 — запрос карточки пользователя в целом (для показа страницы);
- /_back/?type=users&id=12&part=versions — запрос истории редактирования карточки пользователя (только массив изменений и счётчик записей — для AJAX-таблицы);
- /_back/?type=users&part=vocs — запрос набора словарей для показа реестра пользователей (списки подразделений, ролей и т. п.).

Общий формат ответов

Успешные ответы

Все ответы на динамические запросы возвращают либо данные приложения, либо сообщение об ошибке.

Ответы-перенаправления (3xx) на динамические запросы не допускаются. Если в UI необходимо совершить переход на вычислимый URL — эта строка должна приходить в составе JSON-ответа с кодом `200 OK`.

AJAX (JSON)-данные

Для подавляющего большинства запросов ответ должен иметь формат JSON и, соответственно, быть заявлен с `Content-Type:application/json`.

Структура JSON-объектов определяется в основном набором js-библиотек, используемых в UI.

Бинарные данные

Форматы, отличные от JSON, допустимы только при загрузке файлов, типы которых непосредственно оговорены в постановке задачи, а бинарное содержимое может быть вычислено только на сервере. В частности:

- генерируемые документы в формате PDF и подобных ему;
- произвольные файлы, хранимые на сервере, но недоступные для статических запросов из-за необходимости конторлировать доступ на уровне логики приложения.

Для многих видов отчётов (Office XML) существует техническая возможность генерации как на сервере, так и на клиенте. Крайне предпочтительно использовать последний кариант: формировать все возможные файлы при помощи js-процедур по данным, полученным в формате JSON.

Аварийные ответы

Динамические запросы, нарушающие требования аутентификации (например, с просроченным номером сессии) должны получать ответ `401 Unauthorized` — клиент должен распознавать эту ситуацию и реагировать соответственно (показывать форму входа в приложение).

Остальные аварийные ситуации должны оформляться HTTP-ответами типа `application/json`, содержимое которого позволит js-клиенту отобразить необходимое сообщение. Рассматривается 2 вида ошибок, выявляемых сервером:

Исключения, предусмотренные бизнес-логикой

Это такие ситуации, как, например:

- пустое значение обязательного параметра;
- значение параметра, не соответствующее его типу либо выходящее за рамки области допустимых значений;
- нарушение прав доступа в рамках корректно заведённой сессии (например, при обращении по ранее сохранённому URL к странице, недоступной для данного пользователя в текущий момент);
- конфликт с содержимым БД (нарушение уникальности и т. п).

В любом таком случае запрошенное действие не может быть выполнено и причина этого — в данных, присланных с клиента. Сервер же работает корректно. В этом случае:

- код ответа `422 Unprocessable Entity`;
- поле `message`JSON-объекта в ответе содержит человекочитаемое сообщение (либо его код по словарю — если таковой реализован на клиенте);
- поле `field` показывает имя поля ввода, которое js-клиенту следует высветить в качестве проблемного (фокус ввода, красный цвет и т. п.)

Ошибки разработчика и системные сбои

Остальные аварийные ситуации на сервере:

- синтаксические ошибки в прикладном коде (типа некорректно сгенерированного SQL);
- недопустимые операции, которым предшествует недостаточная валидация данных (например: деление на 0 по причине того, что делитель, полученный с клиента, не был проверен на положительность);
- недоступность внешних источников данных (прежде всего, БД) или ошибки в их (сторонних Web-сервисов) реализации;
- исчерпание необходимых вычислительных ресурсов (места на диске);
- прочие сбои ОС и оборудования.

должны оформляться следующим образом:

- код ответа `500 Internal Server Error`;
- поле `id`JSON-объекта в ответе содержит номер (GUID) ошибки, по которому её можно быстро обнаружить в серверном протоколе. UI должен отображать пользователю этот номер в форме, максимально удобной для передачи в службу поддержки (текстовый для копирования в письмо и т. п.)
- поле `dt` — дата в формате `YYYY-MM-DD'T'hh:mm:ss.ms`

Технические детали (текст SQL, содержимое стека) ни показывать пользователю, ни даже передавать с сервера на клиент в каком-либо виде недопустимо.

Клиентская библиотека elu.js

elu.js — это клиентская js-библиотека для разработки web-приложений согласно спецификации elu/dia.

Структура кода

При использовании elu.js пользовательский интерфейс можно (хотя и не обязательно) строить из составных частей, именуемых здесь "блоками".

Функция `show_block` реализует основную связанную с этим операцию: загрузку блока.

Что такое "блоки"

Блоки — это программные единицы приложений на базе elu.js, каждая из которых имеет:

- символическое имя (формат: как идентификатор C, пакетов/пространств имён нет);
- определённое место на экране (обычно прямоугольник, заданный элементом вроде DIV);
- источник данных, которые там отображаются;
- набор действий, связанных именно с этими данными.

Например, если на экране требуется отобразить верхнюю панель с логотипом, часами, ФИО текущего пользователя и кнопкой "Выход" — её можно реализовать в виде блока 'auth_toolbar': поскольку кнопка "Выход" связана с тем же объектом данных ($_USER), что и плашка с ФИО. Тогда процедура отображения общей разметки страницы (сама вызываемая по ходу загрузки блока main), может содержать следующий фрагмент:

```
show_block ('auth_toolbar') // верхняя панель
show_block ('menu')         // боковое меню
show_block ($_REQUEST.type) // что хотел увидеть пользователь, открывая данный URL
```

А вот панель поиска/фильтрации над таблицей сотрудников отдельным блоком делать неразумно: ведь её поля ввода управляют содержимым тела таблицы, а не самой панели.

Как устроен блок

С точки зрения клиентского программного кода, каждый блок — это два или три одноимённых файла, расположенных каждый в своей директории. Два из них содержат js-код, третий (необязательный) является шаблоном HTML5.

Модуль данных: app/js/data/{name}.js

Здесь определяются:

- обработчики действий (см. $_DO);
- источник данных: асинхронную функцию, являющуюся одноимённой блоку компонентой глобального объекта $_GET.

Вот пример части data-модуля блока `user` (страницы учётной записи пользователя системы):

```
$_DO.setpasswd_user = function (e) {
    show_block ('user_setpasswd') // показ другого блока: popup назначения пароля
}
///  ... прочие компоненты $_DO ... ///
$_GET.user = async function (o) {
    let data = await response ({type: 'users'}) // AJAX-запрос /back?type=users?id=...
    data.active_tab = $_LOCAL.get ('user.active_tab') || 'user_options'
      // добавление данных из localStorage
    $('body').data ('data', data) // сохранение переменной в контексте страницы
    return data // выдача для отображения
}
```

Модуль отображения: app/js/view/{name}.js

Здесь определяется функция $_DRAW.(имя_блока), которая, приняв на вход результат вышеописанной $_GET-функции (и больше не вызывая AJAX-запросов), должна отобразить его на экране, сгенерировав необходимые DOM-элементы и назначив им необходимые обработчики событий.

Продолжим наш пример с постейшей страницей учётной записи:

```
$_DRAW.user = async function (data) {
    $('title').text (data.label) // установка заголовка окна браузера

    let $main = $('main').html ( // заполнение требуемого элемента
        (await use.jq ('user'))  // содержимым шаблона,
        .draw_form (data)        // преобразованным API-функцией
    )

    show_block ('user_options')  // отображение вложенного блока
    return $main                 // обозначение обработанной области
}
```

`draw_form` - функция не elu.js, а сторонней библиотеки. Для отрисовки форм и таблиц можно с равным успехом использовать jQueryUI, w2ui и их многочисленные аналоги.

$_DRAW-функция вполне может преобразовать DOM, не возвратив никакого результата. Однако если на выходе у неё jQuery-объект, то:

- всем вложенным элементам приписывается атрибут `data-block-name` с именем данного блока;
- для элементов `<button>` на `click`автоматически назначаются соответствующие по имени функции `$_DO`.

В частности, в нашем примере на

```
    <button name=setpasswd>Установить пароль</button>
```

будет назначена функция `$_DO.setpasswd_user`.

Шаблон: app/html/{name}.html

Как упомянуто выше, $_DRAW-функция преобразует DOM-дерево текущей страницы. Она может делать это напрямую или с использованием разнообразных UI-библиотек.

Если в блоке требуется отобразить готовый фрагмент специфической web-вёрстки, бывает удобно не собирать его поэлементно, а использовать HTML-шаблон, вынесенный в отдельный файл.

Функция use.jq в примере выше позволяет загрузить шаблон из директории `app/html`, получив его содержимое в виде объекта jQuery. Если требуется только исходный текст, можно воспользоваться use.html.

Имя загружаемого файла может быть произвольным, но для упрощения поддержки настоятельно рекомендуется использовать то же имя, что у блока.

Для подстановки данных eju.js содержит функцию to_fill, реализующую разметку HTML5 data-атрибутами, но по своему усмотрению можно использовать jQuery или произвольную стороннюю библиотеку.

Как правило, удобно использовать use.jq и to_fill не напрямую, а через совмещающие их функции-обёртки, отображающие более конкретные типовые элементы дизайна текущего приложения: формы, popup-окна и т. п.

$_DO

Этот js-объект — реестр всех возможных действий в приложении.

В норме его компоненты устанавливаются в файлах `app/js/data`, а ключи имеют вид:

```
{имя_действия}_{имя_блока}
```

Под "действием" понимается js-событие, переименованное в соответствии с логикой предметной области. Допустим, в файле `app/js/data/user.js` есть фрагмент

```
 <button name=delete data-question="Вы уверены?">Удалить</button>
```

Тогда щелчок мышью по этому элементу с точки зрения js является событием `click`, а с точки зрения приложения — действием `delete`.

Соответствующий обработчик должен быть определён в `app/js/data/user.js`. Например, так:

```
$_DO.delete_user = function () {
 query ({type: 'users', action: 'delete'}, {}, function () {
  refreshOpener () // родительскую вкладку — обновить
  window.close ()  // текущую — закрыть
 })
}
```

Далее, если в `app/js/view/user.js` шаблон заполняется функцией fill, этот обработчик будет автоматически обёрнут в confirm с заданным значением data-question и назначен данной кнопке: по $_REQUEST.type и button.name.

У некоторых компонент $_DO названия (специально) не укладываются в описанную схему, они устанавливаются самой elu.js, например, $_DO.apologize.

$_REQUEST

`$_REQUEST` — глобальная переменная, содержащая:

- базовые (имеющие универсальный смысл) параметры отображаемого экрана;
- технические переменные для внутреннего использования API elu.js.

Базовые параметры

Они устанавливаются при загрузке страницы (корневым скриптом, вызываемым requirejs — в норме это /_/handler.js) в зависимости от URL текущего окна.

Эти параметры используются по умолчанию для формирования URL запросов динамического содержимого (dynamicURL), используемых в query и download.

Кроме того, они применяются API при конструировании идентификаторов. В частности, функция fill устанавливает для элементов `button` обработчики click, которые ищет в объекте $_DO по $_REQUEST.type и именам кнопок.

$_REQUEST.type

Символическое имя типа основного объекта, отображаемого на странице.

Например, для списка пользователей системы естественно выбрать `$_REQUEST.type = users` и, соответственно, отображать её по `URI=/users/`. То же символическое имя будет передаваться на сервер в динамических запросах и, если нет специфических затруднений, таблица учётных записей в БД должна бы называться так же.

$_REQUEST.id

Уникальный номер основного объекта, отображаемого на странице.

Например, для пользователя с UUID=52C6B992-3E37-11E7-A5F4-B9194EA6B324 естественно выбрать `URI=/users/52C6B992-3E37-11E7-A5F4-B9194EA6B324` — и это значение UUID стоит присваивать `$_REQUEST.id`.

Внутренние параметры

$_REQUEST._secret

Список имён конфиденциальных полей: их значения передаются query не в теле POST-запроса, а в заголовке. Автоматически устанавливается функцией values.

$_USER

Это глобальная переменная, содержащая данные о текущем пользователе приложения.

В настоящий момент API elu.js никак не использует какие-либо конкретные её компоненты. Желательно только придерживаться следующих правил именования:

- `$_USER.label` — непосредственно отображаемое полное имя пользователя;
- `$_USER.role` — символическое имя его роли (может использоваться для определения видимости кнопок и прочих элементов).

По соображениям безопасности js-приложению не следует знать уникальный номер учётной записи пользователя (где бы она ни хранилась: в реляционной БД, LDAP и т. д.) Вместо id пользователя запросы к серверу приложения должны содержать только (временный, одноразовый) id сессии. И этот id должен передаваться в составе cookie с параметром httpOnly, то есть быть недоступным для js-кода.

Так или иначе, поле *$_USER.id* в клиентском коде использовать не следует никогда.

Проверка наличия сессии

Номер сессии и, возможно, ещё некоторые httpOnly (то есть видимые только серверу) значения — единственное, для чего следует использовать cookies. Современные Web-приложения могут и должны хранить данные от запроса к запросу, используя `localStorage` и `sessionStorage`. Последний, естественно, должен применяться для данных, которые относятся не к приложению в целом (общие справочники), а к текущему сеансу: прежде всего, учётной записи пользователя.

Итак, текущая учётная запись хранится в `localStorage`, в формате JSON, по ключу `user`. Именно таким образом она записывается туда функцией $_SESSION.start.

Извлекается $_USER из `localStorage` автоматически при загрузке elu.js. В норме прикладной код должен переопределять это значение в целом только вызовом $_SESSION.start.

Если текущий пользователь не определён, переменная `$_USER` имеет пустое значение, соответствующая проверка имеет вид

```
if (!$_USER) ...
```

Если логика приложения предполагает обязательную аутентификацию пользователей, то имеет смысл вставить такой if в основной загрузочный скрипт приложения сразу после вызова `setup_user`. И далее:

- для приложений с собственной аутентификацией — должна рисоваться login-форма:

```
if (!$_USER) return use.block ('logon')
```

- для приложений, включённых в ту или иную схему single sign on — должен быть redirect либо query для опознания текущего пользователя сторонними средствами.

Завершение сессии

Информация о пользователе на клиенте стирается вызовом $_SESSION.end. При этом, как правило, следует отправить на сервер соответствующий AJAX-запрос, но не стоит дожидаться и обрабатывать его результат: недоступность сервера не должна блокировать процедуру завершения сеанса на клиенте:

```
$_DO.execute_logout = function () {
  query ({type: 'sessions', action: 'delete'}, {}, $.noop, $.noop)
  $_SESSION.end ()
  redirect ('/')
}
```

Если вы хотите, чтобы при явном завершении сессии все окна (вкладки) браузера, связанные с приложением, оказались закрыты или как минимум очищены, следует воспользоваться функцией $_SESSION.closeAllOnLogout, прописав её на прослушивание localStorage.

Запуск AJAX запросов

Вызов асинхронной функции `await response` запускает AJAX-запрос на JSON-данные и обеспечивает базовую обработку ответа.

Пример

Допустим, $_REQUEST имеет значение:

```
$_REQUEST = {
 type: "stat_form_0609204",
 _secret: ['passwd']
}
```

Тогда вызов

```
await response ({action: 'update'}, {data: {
 name: "v_03_01",
 value: 123,
 passwd: "p4$$w0rd"
}})
```

инициирует POST-запрос на URL со строкой запроса

```
?type=stat_form_0609204&action=update
```

телом

```
{"data":{"name":"v_03_01",value:123}}
```

и дополнительным HTTP-заголовком

```
X-Request-Param-passwd: "p4$$w0rd"
```

Если полученный ответ будет иметь код 200 OK, тип application/json и содержимое

```
{"success":true, "content": {...}}
```

то содержимое `content` будет выдано в качестве результата. При нарушении хотя бы одного из перечисленных условий будет сгенерировано исключение, предварительно обработанное функцией `$_DO.apologize`.

Параметры

1-й параметр — type/id/action/part

1-м параметром передаётся объект для формирования целевого URL функцией dynamicURL.

Он может содержать ключи `type`, `id`, `action` и `part`. Для неупомянутых полей используются одноимённые компоненты $_REQUEST. Если этот параметр опущен, соответственно, URL формируется прямо из $_REQUEST.

Иногда требуется послать AJAX-запрос безо всяких параметров: для поддержания сессии в отсутствие действий пользователя. При этом следует явно указать `{type:undefined}`. Впрочем, для таких запросов предусмотрена специальная API-функция $_SESSION.keepAlive.

2-й параметр — данные запроса

Это объект, содержимое которого будет передано в формате JSON в теле POST-запроса и, частично ($_REQUEST._secret), в HTTP-заголовках. То есть параметры запроса, отличные от базовых `type`, `id`, `action` и `part`.

Если эти данные требуется брать из полей ввода — можно воспользоваться функцией values. Если никаких дополнительных параметров не требуется — параметр может быть опущен.

Обработка ошибок ($_DO.apologize)

Функция $_DO.apologize вызывается await response, если на AJAX-запрос не удаётся получить успешного ответа, и обрабатывает полученное исключение до того, как код приложения получит его в catch-блоке.

Она задаётся в elu.js, однако при разработке приложения её имеет смысл переопределить в учётом местного дизайна и прочих особенностей конкретного проекта. Тем не менее, реализация по умолчанию написана так, чтобы её было удобно если не использовать непосредственно, то брать за основу.

Ниже рассмотрены основные ситуации, обрабатываемые стандартной $_DO.apologize.

401 Unauthorized

Предполагается, что сервер выдаёт ответ с таким статусом, если не может найти сессию по переданным cookies: обычно это бывает при заходе по ссылке из внешнего источника или после перезапуска сервера.

При получении такого ответа очищается sessionStorage (раз сервер не признал сессию — значит, хранить локально нечего) и происходит перезагрузка страницы браузера. Как рисовать страницу при пустой переменной $_USER — решать приложению.

413 Payload Too Large

Такие ответы выдаются для запросов, нарушающих установленный на сервере лимит размера тела HTTP POST. На этот случай предусмотрен фиксированный alert.

504 Gateway Timeout

Фронтальный HTTP-сервер не дождался ответа FastCGI-демона. Клиент перенаправляется на URL=/_maintenance/, где должна быть страница "Извините, ведутся работы". Это можно использовать даже в тех ситуациях, когда работы действительно ведутся: если back-сервер отключён, клиенты будут автоматически направляться на страницу с извинениями.

500 Internal Sever Error, {"id":...}

Фатальные ошибки неизбежны, и когда они случаются, приходится выдавать ответ с кодом 500. Однако важно сделать так, чтобы любую аварию можно было как можно быстрее разобрать, но пользователь узнал бы при этом минимум деталей.

Предполагается, что любой 500-нный HTTP-ответ имеет корректное JSON-тело с компонентой id. При получении такового показывается alert с просьбой записать id и обратиться в поддержку. Разумеется, выдавая этот ответ, сервер сначала должен записать в error.log этот id (случайный и подлиннее), текущее время и все известные детали ошибки.

{"message":..., "field":...}

Хоть бы и 200-й статус с корректным JSON — ещё не гарантия того, что всё хорошо. Таковой гарантией является истинное значение поля `success`.

Если же истинного success не наблюдается, но есть поле `message`, то его значение отобразится в alert. А если при этом ещё имеется компонента `field` и по ее значению на текущем экране удастся найти поле ввода, то такое поле получит фокус.

Ответы описанного вида предусмотрены для высокоуровневых ошибок, выявляемых при проверке данных, прав доступа и т. п.

Получение бинарных файлов (download)

POST-запрос на получение файла для скачивания (его имя должно быть в заголовке ответа Content-Disposition).

Пример

Допустим, $_REQUEST имеет значение:

```
$_REQUEST = {
 type: "stat_form_0609204",
 id: "E849AE72-3E17-11E7-89B6-B9174EA6B324"
}
```

Тогда код

```
function label (cur, max) {return String (Math.round (100 * cur / max)) + '%'};
w2utils.lock (box, label (0, 1));
download (
```

```
 {action: 'download_csv'},
 {name: '03_5'},
 {
  onprogress: function (cur, max) {$('.w2ui-lock-msg').html ('

' + label (cur, max))},
  onload:     function () {w2utils.unlock (box)},
 }
)
```

покажет прогресс-индикатор (w2utils.lock) и инициирует POST-запрос на URL со строкой запроса

```
?type=stat_form_0609204&id=E849AE72-3E17-11E7-89B6-B9174EA6B324&action=download_csv
```

и телом

```
{"name":"03_5"}
```

Далее по ходу получения тела файла он будет показывать прогресс (onprogress), а по окончании процесса скроет индикатор (onload).

Параметры

Первые два -- такие же, как у query.

Третий -- набор callback'ов: onprogress и onload. Отметим, что onload вызывается по окончании передачи данных в любом случае: в том числе и аварийном. Поскольку зависший прогресс-индикатор не нужен никому.

Результат

Если на запрос приходит успешный (200 OK) ответ, то пользователь получает диалог открытия/сохранения файла с именем из Content-Disposition.

В противном случае выдаётся alert с сообщением об ошибке.

Отправка бинарных файлов (Base64file.upload)

Для передачи на сервер бинарных файлов произвольной длины предусмотрена функция `Base64file.upload`.

Её вызов инициирует серию AJAX-запросов.

```
        var data = {
            the_type     : $_REQUEST.type,
            the_id       : $_REQUEST.id,
            note         : r.note,
            id_file_type : r.file_type.id,
        }

        var n = r.files.length

        var check = setInterval (function () {
            if (n) return
            clearInterval (check)
            $_DO.apologize = $.noop
            location.reload ()
        }, 100)

        var opt = {
            data       : data,
            type       : 'contract_files',
 //         action     : {create: 'create', update: 'update'},
            onprogress : function (x, y) {sum += portion; $progress.val (sum)},
            onloadend  : function () {n --}
        }

        $.each (r.files, function () {Base64file.upload (this.file, opt)})
```

При передаче файла производится минимум 2 await response:

- первый — с `action=opt.action.create`;
- второй и последующие — с `action=opt.action.update`.

Если `type` соответствует такой бизнес-сущности, как "файл", "вложение" и т. п., удобно не указывать опцию `action`, используя значения по умолчанию. Но для объектов вроде "договор" даже при условии уникальности файлового вложения может понадобиться развести операции со скалярными полями и с бинарным содержимым по разным действиям, прописав, например, `action: {update: 'append'}`.

Значение параметра `type` у всех запросов совпадает (`opt.type`). Остальные — зависят от типа запроса (см. ниже).

Запрос opt.action.create

Первый запрос, с `action=opt.action.create`, предназначен для создания на сервере временного буфера, пригодного для дальнейшего приёма данных.

Из содержимого файла здесь не передаётся ни байта — только хэш-параметр `file`, где содержатся его:

- имя (`label`);
- длина (`size`);
- MIME-тип (`type`);

а также все прочие компоненты, полученные Base64file.upload в составе объекта `data`.

Получив такой запрос, сервер может отказать в приёме файла (подробнее об оформлении ошибок валидации см. response).

Если же ошибки не возникло и временный буфер создан — серверный обработчик должен вернуть:

- либо его уникальный id как скаляр;
- либо объект с соответствующим полем `id`.

Примерный вид серверного обработчика:

```
sub do_create_contract_files {
  my $file = $_REQUEST {file};
  $file -> {uuid} = _new_uuid ();
  $file -> {id}   = sql_do_insert (contract_files => $file);
  return $file -> {uuid};          # либо return {id => $file -> {uuid}};
}
```

Запрос opt.action.update

Запросы на передачу тела файла содержат следующие параметры:

- `type` (тот, который указан при вызове Base64file.upload в `opt.file`);
- `id` (тот, что получен в ответе на запрос `opt.action.create`);
- `action` (значение `opt.action.update`);
- `chunk` — часть тела файла, закодированная в Base64.

Длина `chunk` определяется `opt.portion`, по умолчанию это 128 Кб.

Каждый последующий запрос выполняется только после получения ответа на предыдущий, так что на сервере можно приписывать каждый отрезок в конец временного файла.

При получении ответа на каждый запрос `Base64file.upload` вызывает функцию `opt.onprogress` (если она определена) — для индикации состояния процесса.

По завершении процедуры вызывается `opt.onloadend`.

Последний update-запрос никак специально не помечается для сервера: имеется в виду, что тот в любом случае должен отслеживать переданный объём и завершать операцию, когда размер накопленного файла совпадёт с изначально заявленным размером.

Примерный вид серверного обработчика:

```
sub do_update_contract_files {

  my $data = sql (contract_files => [
      [fake  => $_REQUEST {sid}],
      [uuid  => $_REQUEST {id}],
      [LIMIT => 1],
  ]); $data -> {id} or die "contract_file $_REQUEST{id} not found";

  my $chunk = MIME::Base64::decode ($_REQUEST {chunk});
  my $tmpfn = "/tmp/$data->{uuid}";
  my $old = 0 + -s $tmpfn;
  my $new = length $chunk;
  my $on = $old + $new; $on <= $data -> {file_size} or die "$on bytes sent but only $data->
{file_size} bytes claimed";

  open (F, ">>$tmpfn") or die "Can't append to $tmpfn:$!\n";
  binmode F;
  print F $chunk;
  close (F);

  return if -s $tmpfn < $data -> {file_size};

  my ($y, $m) = Date::Calc::Today ();

  my $path = sprintf ('%04d/%02d', $y, $m);
  my $abs_path = $preconf -> {files} . '/' . $path;
  $data -> {file_name} =~ /\.\w+$/;
  my $ext = $&;

  File::Path::make_path ($abs_path);
  $data -> {file_path} = "$path/$data->{uuid}$ext";
  File::Copy::move ($tmpfn, "$preconf->{files}/$data->{file_path}");

  $data -> {fake} = 0;
  delete $data -> {contract};
  sql_do_update (contract_files => $data);

}
```

Запрос без ответа (jerk)

Для запуска долгих процедур, результата которых не имеет смысла ожидать ранее, чем через несколько секунд, предусмотрен специальный аналог response: `jerk`.

В такой ситуации, когда ожидание долее 10 секунд нормально, ни в коем случае не следует использовать query: он выдаст ошибку при живом процессе.

Увеличивать timeout -- не вариант, поскольку пользователь, долго глядя на одно и то же (хоть бы и вращающееся одно и то же), всё равно почувствует, что программа "зависла". Даже если это и не так.

Необходимо поступить ровно противоположным образом: урезать timeout до нуля, гарантировав получение соответствующей ошибки (это и делает jerk) -- чтобы далее периодически опрашивать сервер о текущем состоянии дел (обычным await response).

Если await response направляет свои запросы на URI `/_back`, то jerk -- на `/_back/_slow`. Это сделано для того, чтобы избежать ожидания не только на стороне клиента, но и proxy-сервером.

Пример

Здесь считается, что для текущего $_REQUEST.type реализованы 2 запроса:

- с action=import -- запуск процедуры
- с part=import -- опрос текущего состояния.

Последний возвращает структуру данных, у которой значение поля label по ходу процедуры описывает текущий прогресс (например, в процентах), а когда процесс прекращается, оно становится пустым.

```
       jerk ({action: 'import'}, {}, function () {

           var data = {label: 'Импорт...'}

           wait ({

               interval: 1000,

               until: function () {
                   if (!data.label) return true
                   $('.progress').text (data.label)
                   query ({part: 'import'}, {}, function (d) {data = d}
               },

               then: function () {grid.reload ()},

           })

       })
```

Соответствующий фрагмент app.conf:

```
location /_back {

   gzip on;
   gzip_http_version 1.0;
   gzip_types application/json application/vnd.ms-excel;

   rewrite /_back/? / break;

   fastcgi_pass   my_fcgi;
   fastcgi_connect_timeout 100ms;
   fastcgi_ignore_client_abort on;

   include        fastcgi_params;

   access_log /$document_root/../../back/logs/rq.log Dia.pm.my;

}

location /_back/_slow {

   rewrite /_back/_slow/? / break;

   fastcgi_pass   my_fcgi;
   fastcgi_connect_timeout 100ms;
   fastcgi_read_timeout 1ms;

   include        fastcgi_params;

   access_log /$document_root/../../../back/logs/rq.log Dia.pm.my;

}
```

Отметим, что у `/_back/_slow`:

- минимальный `fastcgi_read_timeout`, что гарантирует моментальную выдачу 504 Gateway Time-out на любой запрос;
- отсутствуют настройки gzip: сжимать тут нечего.

То, что `fastcgi_pass` общий -- простой частный случай. Долгие операции вполне могут быть вынесены и на отдельный FastCGI-сервер.

Отображение результатов (fill)

Асинхронный вызов `await to_fill (name, data[, target])` загружает HTML5-шаблон с указанным именем из `app/html`, заполняет его предоставленными данными и выводит результат в заданную область экрана.

Исходные данные сохраняются в контексте корневых DOM-элементов результата функцией jQuery.data().

Шаблон представляет собой стандартный документ HTML5, без каких-либо синтаксических расширений. Привязка данных к элементам производится на основании значений атрибутов. Кроме того, объект данных может содержать специальные компоненты, влияющие на интерпретацию шаблона (их имена начинаются с символа подчёркивания: `_read_only`, `_can`).

Пример

Допустим, приложение содержит файл `app/html/my_block.html` с содержимым

```
 <div
   data-text="org.label"
   data-id-field="org.id"
   class="official" />
```

`data` — объект с данными:

```
{org: {id: 1, label: "Snake Oil Inc."}}
```

, и в текущем окне браузера есть единственный элемент `main`, содержащий `article`:

```
<main>
 <article>  ...waiting...  </article>
</main>
```

Тогда, если по ходу `$_DRAW.my_block` произвести вызов

```
await to_fill ('my_block', data, $('main > article'))
```

, то `main` примет вид

```
  <main>
   <article>
    <div data-block-name="my_block" class="official" data-id="1">Snake Oil Inc.</div>
   </article>
  </main>
```

и исходный объект данных впоследствии можно получить вызовом

```
$('div[data-block-name=my_block]').data ('data')
```

Правила интерполяции

Первое, что делает `to_fill` — загружает (функцией use.jq) шаблон по имени, заданным первым параметром. Предполагается, что оно должно дублировать имя блока. Не исключён вариант использования нескольких шаблонов в одном блоке — тогда их имена должны быть связаны очевидным образом (имя блока — общий префикс). В принципе никто не мешает разработчику использовать произвольные имена шаблонов. Как он далее будет поддерживать проект — его забота.

Элементы форм

В этом разделе описаны правила интерполяции для HTML-элементов, традиционно относимых к внутренности тегов `form`. Правда, их использование вне этих контейнеров не является ошибкой, а в шаблонах elu.js `form` как таковые не нужны вовсе.

Поля ввода (input/textarea/select)

Элементы `input`, `textarea` и `select` привязываются к данным очевидным образом: по имени (атрибут `name`). Для всех возможных полей вызывается `jQuery.val(...)`, куда передаётся одноимённое значение: например, для [ ] устанавливается значение `data.label`.

В каждой группе полей типа `radio` для 1-го элемента, подходящего по значению, устанавливается свойство`checked`.

Для полей типа `select`:

- если у первой опции отсутствует атрибут `value`, то приписывается `value=""` и эта опция выбирается для значений `null`;
- для опций с `data-list` по умолчанию принимается `data-value="id" data-text="label"`.

Таким образом, типовой drop down для словаря стандартного вида (id/label) оформляется как

```
<select name=id_role>
 <option value="">[не назначена]</option>
 <option data-list=roles.items>
</select>
```

._fields: схема данных

Помимо значений, `to_fill` расставляет в `input` и `textarea` дополнительные атрибуты на основании описания схемы данных, присланной с сервера. Для этого объект должен содержать поле `_fields` следующего вида:

```
...
my_field: "some_value",
other_field: null,
_fields: [
 my_field: {
  TYPE:   "...", // влияет на type, если он не указан
  REMARK: "...", // пока не используется
  COLUMN_SIZE:   // -> атрибут maxlength
  MIN_LENGTH:    // -> атрибут minlength
  MIN:           // -> атрибут min
  MAX:           // -> атрибут max
  PATTERN:       // -> атрибут pattern
 },...
]
```

Разрешение неоднозначности

Атрибуты, изначально указанные в HTML-шаблоне, имеют более высокий приоритет: например, если с сервера пришло `MIN:"1970-01-01"`, а в самом input'е прописано `min="2000-01-01"`, то последнее останется без изменения. Имеется в виду, что сервер показывает габаритные ограничения допустимых областей, а на конкретных экранных формах они могут уточняться.

Уточнение типа поля

Явное указание `<input type="..."` не переопределяется.

Однако если атрибут `type` отсутствует, а с сервера пришло `TYPE: "date"`, то в DOM будет установлено `<input type="date"...`.

Для значений `TYPE`, начинающихся на `int` и `num`, устанавливается [ ]. При этом одновременно:

- если нигде не задана опция `min`, то принимается `min="0"`;
- положительное значение `DECIMAL_DIGITS` генерирует `step="0.00...01"`;
- значение `maxlength` корректируется в зависимости от того, нужно ли резервировать место под:
  - знак '-' (если `min < 0`);

- десятичную запятую (если `step` начинается на `0.`).

Повторим: все перечисленные эвристики применяются лишь когда соответствующие опции не определены. Если же, скажем, требуется описать поле ввода температуры воздуха где-то на уровне моря, никто не мешает явно прописать, например, `min=-60 step=5 maxlength=5` или вообще `type=text`. Но по умолчанию для числовых реквизитов будут генерироваться поля, не допускающие отрицательных значений — что обычно и требуется в учётно-аналитических системах.

Замечание о семантике

Наличие атрибутов не определяет их дальнейшей обработки.

Лишь ограничение `maxlength`, введённое ещё в HTML 3.2, гарантируется браузером просто и очевидно: по достижении указанной длины символы просто перестают добавляться. Даже при отключённом js. Правда, и тут не обошлось без фокусов: это работает для текстовых полей, к которым не относится, например, новый `type="number"`.

Кажущийся симметричным атрибут `minlength` по очевидным причинам столь же жёстко, на этапе ввода, работать не может. Да и вообще о нём мало известно.

Атрибут `pattern` непосредственно управляет только CSS-псевдоклассами `:valid`/`:invalid`. А `min` и `max`сами по себе вообще должны игнорироваться для формально неподходящих типов полей: то есть, например, для поля ввода даты, изготовленного из `type="text"` добавлением js-календаря.

Однако полный автоматический учёт всего перечисленного реализован в результате другой функции: values.

Подстановка меток полей

HTML5 устанавливает два способа связать элемент `LABEL` с соответствующим полем ввода:

- атрибутом `for="..."`;
- включением поля в `LABEL`.

Обоими этими способами можно воспользоваться, чтобы не писать имя поля на экранной форме, если оно не отличается от заданного ещё в схеме БД комментария (`data._fields [input.name].REMARK`):

```
<label for=id_voc_org /><select name=id_voc_org>
  <option data-list=voc_orgs.items data-text=label data-value=id>
</select>
<label> <! -- будет подставлен атрибут for=position -- >
  <input name=position />
</label>
```

Подстановка выполняется только для пустых `LABEL`: как и в случае с атрибутами валидации, прямое указание содержимого отключает всю автомагию.

._read_only: режим просмотра

Если у объекта данных истинно поле `_read_only`, все поля ввода, кроме `hidden`, заменяются на span'ы с соответствующим текстом. Например, для `data = {_read_only: 1, label: "foo"}` шаблон

```
<input name=label>
```

превращается в

```
<span>foo</span>
```

Для каждой группы полей типа `radio` все элементы, не подходящие по значению, удаляются каждый со своим непосредственным родителем, а подходящие — просто удаляются. Например, для `data = {_read_only: 1, id_status: 1}` шаблон

```
<div><input name=id_status value=1>New</div>
<div><input name=id_status value=2>In Progress</div>
<div><input name=id_status value=3>Terminated</div>
```

превращается в

```
<div>New</div>
```

Кнопки (button)

Назначение обработчиков

С точки зрения HTML5-вёрстки для изображения прямоугольника с надписью "Утвердить" у тега `button` нет никаких преимуществ перед `div` прочими block-элементами. Скорее наоборот: стандартное браузерное отображение обычно не вписывается в требуемый дизайн и его надо специально подавлять в CSS.

Однако в elu.js рекомендуется изображать кнопки именно как `button`. Тогда функция `to_fill`, вызванная по ходу use.block автоматически будет подключать обработчики события `click`, находя их в переменной $_DO. Например, внутри вызова `use.block ('stat_form_0609204')` для

```
<button name=accept>Утвердить</button>
```

на клик будет назначена функция `$_DO.accept_stat_form_0609204` и параллельно установлен класс `clickable` (см. clickOn). Если же значение `$_DO.accept_stat_form_0609204` не определено, то кнопка останется без обработчика и без `clickable`.

Имя текущего блока определяется по атрибуту `data-block-name` коревого элемента шаблона (а устанавливает его там use.block).

Обработка исключений

Функция `to_fill` назначает кнопкам компоненты $_DO не непосредственно, а через дополнительную обёртку, обрабатывающую исключения.

Если $_DO-процедура выбрасывает в качестве исключения строку специального вида:

```
#{имя поля}#:{текст сообщения}
```

то устанавливается фокус на поле с заданным именем и вызывается `alert` с соответствующим текстом сообщения. Этот механизм предусмотрен для валидации:

```
$_DO.update_user = function () {

    var data = values ($('.drw.form'))

    if (data.password) {
        if (!data.password2) throw '#password2#:Для страховки от опечаток необходимо ввести пароль повторно'
        if (data.password != data.password2) {
            $('input[type=password]').val ('')
            throw '#password#:Ошибка при 1-м или 2-м вводе пароля'
        }
    }
    ...
}
```

.can: управление видимостью

Большинство форм содержат наборы кнопок, каждая из которых должна показываться не всегда, а в зависимости от многих факторов (прав текущего пользователя, статуса отображаемого объекта и т. п).

Показом/сокрытием кнопок можно управлять, как у для любых других элементов: атрибутами `data-off` / `data-on` (см. ниже).

Однако для элементов `button` функция `to_fill` поддерживает ещё один аналогичный механизм, связанный с полем `data._can`. Если такое поле определено, то оно должно быть объектом, содержащим истинные значения для имени каждой кнопки, которую требуется показать. Например, при

```
data: {_can: {
 print: 1,
 accept: true,
 kick: null
}}
```

шаблон

```
<button name=print>Печать</button>
<button name=publish>Отправить ОМСУ</button>
<button name=kick>Отклонить</button>
<button name=accept>Принять</button>
```

превратится в

```
<button name=print>Печать</button>
<button name=accept>Принять</button>
```

data-question: уточняющий вопрос

Для многих действий обработка начинается с переспроса о том, насколько уверен пользователь. Это можно реализовать в обработчике click, но есть и другой вариант: прописать вопрос прямо в HTML-шаблон:

```
<button name=accept data-question="Вы уверены, что хотите утвердить?">Утвердить</button>
```

Тогда по ходу интерполяции функция `to_fill` (точнее, вызываемая ей clickOn) добавит в начало обработчика соответствующий `if (confirm ()) ...`.

Неименованные элементы: data-атрибуты

Для HTML-элементов, не имеющих (в отличие от полей форм) стандартной привязки к данным, ассоциация устанавливается при помощи дополнительных атрибутов с префиксом `data-`.

Кроме того, некоторые data-атрибуты выполняют роль управляющих конструкций (типа if и for).

data-text / data-html: вставка текста

Атрибут `data-text` привязывает поле данных к тегу HTML так же, как `input/name`: внутри элемента отображается соответствующий текст. Осторожно: все дочерние элементы при этом удаляются, так что тег с data-text в шаблоне должен быть листом DOM-дерева.

Атрибут `data-html` действует аналогично, но подставляет в родной тег не плоский текст, а HTML-поддерево. Для `data = {the_text:"Perl6 is **out**!"}` шаблон

```
<article data-html="the_text" />
<div id="html_source" data-text="the_text" />
```

превращается в

```
<article>Perl6 is <b>out</b>!</article>
<div id="html_source">Perl6 is &lt;b&gt;out&lt;/b&gt;!</div>
```

Форматирование содержимого для data-text: data-default, data-digits

Значение по умолчанию: data-default

По умолчанию значения `null` и `undefined` при подстановке заменяются на строку нулевой длины — и элемент с соответствующим `data-text` получается пустым.

Если в данном фрагменте шаблона требуется заменять все пустые значения на некоторую константу, для этого можно воспользоваться атрибутом `data-default`:

```
Оплачено: <div data-text=dt_payment data-default="пока что нет..."></div>
```

Постановка из словаря: data-voc

Если

- одно из полей (например, `data.id_role`) запись содержит ключ строки в некоем словаре,
- а другое (например, `data.roles`) — сам словарь,

отобразить соответствующую строку можно вот так:

```
<span data-text="id_role" data-voc="roles" />
```

Форматирование чисел: data-digits

Для форматирования чисел предусмотрен атрибут `data-digits`. Если его значение — натуральное число, то значение при выводе превращается в js Number, а потом форматируется методом `toLocaleString` с соответствующим числом десятичных знаков:

```
Цена: <div data-text=amount data-digits=2></div>
<! -- "1002.75" => "1 002,75" для locale = ru -- >
```

Нечисловые значения при этом рассматриваются как пустые и не отображаются. Часто, хотя и не всегда, рядом с `data-digits` имеет смысл поставить `data-default=0`.

Пустое значение атрибута `data-digits` приравнивается к нулевому и может применяться для форматирования целых чисел:

```
Число заходов: <div data-digits data-text=count></div>
```

Форматирование дат: data-date

Атрибут `data-date` без значения задаёт форматирование значений методом Date.toLocaleDateString.

```
Дата рождения: <div data-date data-text=dt_birth></div>
```

Само значение при этом может быть в формате 'YYYY-MM-DD' или 'ДД.ММ.ГГГГ' с произвольными разделителями и, возможно, постфиксом времени. В любом случае используются только первые 10 символов.

Есть возможность указать набор опций `toLocaleDateString` — в качестве значения атрибута, в формате JSON:

```
Нынче у нас: <div data-date='{"month":"long"}' data-text=dt_now></div>
```

Форматирование дат со временем: data-ts

Если помимо даты требуется указать точное время, вместо `data-date` следует воспользоваться аналогичным атрибутом `data-ts`, соответствующим Date.toLocaleString.

В отличие от `data-date`, данные для `data-ts` должны соответствовать формату ISO 8601.

```
<div>Данные по состоянию на <span data-text=ts_stats data-ts></div>
```

data-img: вставка графики

Если одно из полей объекта данных содержит изображение (внимание: не HTTP-ссылку на изображение, а именно тело PNG- или JPEG- файла, закодированное в Base64), то его можно назначить фоном элемента, прописав ему соответствующий атрибут `data-img`. Для `data = {photo = "..."}`

```
<div data-img="photo" />
```

превращается в

```
<div style="background-image:url(data:...)" />
```

data-src: определение источника

Можно размещать на странице связанные с данными изображения и более традиционным способом: в виде отдельных файлов-картинок. Лучше всего это подходит для иллюстраций к новостям и прочим изображениям, не требующим проверки прав доступа:

```
<article>
 <img data-on=is_illustrated data-src=img_url align=right width=33%>
 <span data-html=html_body />
 <br clear=all>
</article>
```

data-class: установка CSS-класса

Значение соответствующей компоненты данных передаётся в вызов `jQuery.addClass()`. Например, для `data =`

```
{label: "My Task", status: {label: "expired"}}
```

```
<div data-text="label" data-class="status.label"/>
```

превращается в

```
<div class="expired">My Task</div>
```

data-uri: подключение гиперссылки

В elu.js в принципе не предполагается задание гиперссылок при помощи HTML-тегов `a`. Визуально ссылки должны размечаться при помощи CSS, а HTTP-запросы — инициироваться js.

Для установки URL, связанного с конкретным элементом, предусмотрен атрибут `data-href`. Он полностью аналогичен стандартному `href`, но, в отличие от него, может легально присутствовать не только в `a`, но и в любых других элементах.

А чтобы подставить в `data-href` значение из объекта данных, функция `to_fill` использует атрибут `data-uri`. При этом каждый текстовый узел внутри элемента оборачивается в дополнительный `span` фиксированного класса `anchor`. Например, для `data = {id:1, label="Проект 1", uri="/projects/1/"}`

```
<tr data-uri="uri"/>
 <td data-text="id"/>
```

```
  <td data-text="label"/>
 </tr>
```

превращается в

```
 <tr data-href="/projects/1/"/>
  <td> <span class=anchor>1</span> </td>
  <td> <span class=anchor>Проект 1</span> </td>
 </tr>
```

Дополнительная обёртка позволяет определять собственные CSS-классы и js-обработчики для текстового содержимого, не смешивая их с CSS/js охватывающих контейнеров. Клик по строке "Проект 1" может приводить к открытию вкладки /projects/1/, а клик по клетке таблицы вне этого текста — к чему-то другому (например, выделению самой клетки).

data-id-field: установка id

Этот атрибут имеет смысл использовать, если требуется запомнить id объекта в контексте HTML-элемента не для перехода по ссылке, а для каких-то других действий. При `data = {id: 123, label: "My Task"}`

```
 <div data-text="label" data-id-field="id"/>
```

превращается в

```
 <div data-id="123">My Task</div>
```

data-name: имя поля ввода

Некоторые поля ввода (в основном checkbox'ы в группах) генерируются по выборкам данных и, соответственно, должны иметь вычислимые имена:

```
 <checkbox data-name=uuid>
```

превращается в

```
 <checkbox name="32458aaa-b575-4841-aea1-80eda0f8004d">
```

data-off / data-on / data-roles: выключатели

Элементы с атрибутами data-off / data-on могут быть полностью скрыты (jQuery.hide ()) в зависимости от значения одноимённого поля данных: соответственно, при его истинном значении и в противном случае.

Не-boolean значения трактуются в соответствии с ToBoolean, за одним исключением:

- строка `"0"` считается не истинным (в ECMA-262 все строки ненулевой длины как бы true), а ложным значением — соответственно, для нулевого целого значения, пришедшего в качестве JSON-строки, элементы с `data-off` будут показаны, а с `data-on` — скрыты.

Например, из следующих двух tr шаблона:

```
  <tr data-off="_read_only">
   <th>Отчество:</th>
   ...
  </tr>
  <tr data-on="_read_only">
   <th>ФИО:</th>
   ...
  </tr>
```

в результате будет отображён только один, в зависимости от data._read_only.

Довольно часто условие показа элемента сводится к проверке роли пользователя ($_USER.role) на принадлежность к фиксированному списку. На этот случай предусмотрен специальный атрибут: `data-roles`.

data-list: аналог map/for-each

Если для элемента прописан атрибут data-list, то соответствующее поле данных должно быть списком, а в результате `to_fill` данный элемент тиражируется и интерполируется при помощи той же `to_fill`. Например, для

```
 data = {stages: [
  {uri="/stages/1", label="Первый этап", status: {label: "ok"}},
  {uri="/stages/2", label="Второй этап", status: {label: "fail"}},
 ]}
```

шаблон

```
 <table class="list">
   <tr data-list="stages" data-uri="uri">
    <td data-text="label" data-class="status.label"></td>
   </tr>
 </table>
```

превращается в

```
 <table class="list">
   <tr data-href="/stages/1">
    <td class="ok"><span class=anchor>Первый этап</span></td>
   </tr>
   <tr data-href="/stages/2">
    <td class="fail"><span class=anchor>Второй этап</span></td>
   </tr>
 </table>
```

Обновление области (refill)

Область экрана, заполненная посредством `await to_fill`, запоминает применённый шаблон.

Если требуется обновить её, заполнив тот же новыми данными, можно применить функцию `refill`:

```
  refill (new_data, $('.button-area', $form_container))
```

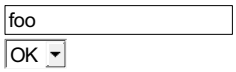
Получение введённых данных (values)

values

Вспомогательная функция, предназначенная для сбора данных с полей ввода (input и select) при запуске query и download.

Пример

Допустим, на экране отрисован HTML:



Тогда вызов

```
query ({}, {
 search:
  values ($('.toolbar'))
}, done)
```

инициирует AJAX-запрос (подробнее см. query), в POST-теле которого будет передано

```
{"q":"foo","status":"OK"}
```

а корректный JSON-ответ будет обработан функцией done().

Параметр

Единственный параметр — объект jQuery, содержащий (на любых уровнях вложенности) поля, которые требуется передать.

Результат

Функция values возвращает js-объект, ключами которого являются имена (атрибут name) полей `input` (кроме `type=password`), `textarea` и `select`.

Значениями для всех полей, кроме `input type=checkbox`, является то, что возвращает для них функция jQuery.val().

Что касается `checkbox`'ов — они обрабатываются по-разному:

- `checkbox` без атрибута `value` считается скалярным полем (переключателем — поле с таким `name` на форме должно быть только одно), ему соответствует числовое значение `0` либо `1`;
- `checkbox` с атрибутом `value` воспринимается как инструмент множественного выбора, ему всегда соответствует массив, состоящий из значений `value` выбранных (`checked`) полей. Если ни один `checkbox` с заданным `name` не выбран, для него возвращается пустой массив.

И ещё его прототип имеет пару методов зачистки/проверки данных, о которых чуть ниже.

$_REQUEST._secret: передача конфиденциальных данных

Имена полей типа `password` собираются в массив `$_REQUEST._secret`. Вызов await response (query) передаёт упомянутые там поля не в JSON-теле POST-запроса, а в дополнительных HTTP-заголовках X-Request-Param-..., что позволяет скрыть их от автоматического логирования.

actual (): игнорирование скрытых полей

У сложных форм ввод значений в одни поля (в основном `select` и `radio`) может приводить к сокрытию или, наоборот, показу фрагментов, содержащих другие поля.

Например: переключатель "физическое/юридическое лицо" открывает раздел с соответствующими реквизитами (методом вроде $.slideAsNeeded).

Значения полей, убранных из видимой области, обычно передаваться на сервер не должны. Однако по ходу переключений там вполне может образовываться мусор. Для того, чтобы от него избавиться, предусмотрен метод

```
values ().actual ()
```

Он выдаёт тот же объект, только предварительно удаляет из него значения всех убранных полей. Поля типа `hidden` остаются без изменения.

validated (): проверка значений

Этот метод вызывается ради побочного эффекта: если он находит поле, значение которого не попадает в допустимую область, то фокусирует на нём ввод и показывает сообщение об ошибке (die).

Область допустимых значений определяется атрибутами:

- `required`;
- `maxlength`;
- `minlength`;
- `min`;
- `max`;
- `pattern`.

Все атрибуты, кроме `required`, принимаются в расчёт только для непустых значений. Например, пустая строка подходит и для `minlength=3`, и для `pattern=^\d\d\d$`, а строка "00" -- не подходит. Пустое значение отсеивается только атрибутом `required`.

В действительности все проверки производятся ещё при создании результата `values`. Вызов `validated ()` лишь проверяет приготовленный список. И

список этот может быть отфильтрован предшествующим вызовом `actual ()`. Что позволяет реализовывать условно обязательные поля: например, СНИЛС обязателен для физического лица (когда он только и виден), а ОГРН -- для юридического.

Поэтому набор данных с формы рекомендуется получать следующим образом:

```
values ().actual ().validated ()
```

Переходник для w2ui

впрыск введённых значений в AJAX запрос на источник данных для списочного поля формы расширенного поиска (w2injectSearchValues)

загрузка на сервер файлов с popup формы (w2_upload_files_from_popup)

Данная асинхронная функция реализует следующий сценарий:

- находит на экране w2ui popup
  - в нём — форму ввода
    - там — первое поле типа `file`
- подсчитывает суммарную длину содержащихся там файлов;
- визуально оформляет отправку:
  - блокирует форму;
  - скрывает кнопки;
  - показывает элемент `progress`;
- отправляет все файлы методом Base64file.upload
  - используя переданные параметры;
  - отмечая процент загрузки на `progress`.

Перед отправкой всем файлам с неопределённым MIME-типом проставляется `application/octet-stream`.

По окончании процедуры popup остаётся заблокированным: как именно обновлять экран — дело вызывающего блока.

Результат, доступный для получения посредством await -- список id файлов (то, что вернули create-запросы и было использовано по ходу update-запросов).

Пример:

```
var v = w2_popup_form ().values ()
if (v.number_  == null) die ('number_', 'Укажите, пожалуйста, номер документа')
if (v.date_    == null) die ('date_', 'Укажите, пожалуйста, дату документа')
// ... прочие проверки
let data = await response ({type: 'precepts', action: 'create', id: null}, {data: v})
let ids = await w2_upload_files_from_popup ({type: 'precept_files', data: {uuid_precept: data.id}})
w2_close_popup_reload_grid ()
```

закрытие popup с обновлением таблицы (w2_close_popup_reload_grid)

Эта функция реализует типовое действие в конце обработки popup-окна, вызванного из таблицы:

- закрывает окно
- обновляет содержимое таблицы.

```
w2_close_popup_reload_grid ()
```

Таблица на экране должна быть одна (поскольку используется w2_first_grid).

Если в результате операции создаётся новая запись, имеющая обственную страницу, рекомендуется дополнительно открыть её при помощи w2_confirm_open_tab.

обнаружение и разблокировка панели со значком ожидания (w2_waiting_panel)

При использовании elu с w2ui каждый блок загружается, как правило, либо в popup, либо на панель layout'а.

В последнем случае, как положено в AJAX-интерфейсах, панель желательно предварительно очистить, затемнить и повесить на неё индикатор ожидания. Визуальная часть этой работы выполняется в `w2layout.lock`, вызываемом автоматически из-под обработчика, назначенного w2relayout.

```
$panel.w2relayout ({
    name: 'my_layout',
    panels: [
        {type: 'main', size: 800,
            tabs: [
                {id: 'my_layout_block_first_tab', caption: '...'},
                //...
            ].filter (not_off),
        },
    ],
})
```

Итак, `my_layout_block_first_tab` будет загружен в панель, видимую на экране и содержащую индикатор загрузки, который установил предшествующий `lock ()`. Соответственно, презентационная часть блока должна убрать затемнение и заполнить панель своим содержимым. Для этого можно найти в `w2ui` layout и панель явно по именам — тогда необходимо прописывать их в коде.

А можно воспользоваться `w2_waiting_panel`, которая найдёт панель как раз по индикатору ожидания, подготовит её и выдаст соответствующий `DIV`:

```
$_DRAW.my_layout_block_first_tab= async function (data) {
  $(w2_waiting_panel ()).w2regrid ({
  ...// это типовой вариант для таблиц

$_DRAW.my_layout_block_first_tab= async function (data) {
  var $panel = $(w2_waiting_panel ())
  await to_fill ('my_layout_block_first_tab', data.item, $panel)
```

```
$panel.w2reform ({
...// а это — для форм ввода
```

обработка словарей (add_vocabularies)

Это клиентский аналог серверной add_vocabularies.

Функция предназначена для перевода предварительно загружаемых наборов справочных записей:

- из формата словарей Dia.pm (id/label)
- в формат списочных полей w2ui (id/text)

и их дополнительной индексации.

Пример:

```
            add_vocabularies (data, {
                voc_commissioning_act_status: {},
                voc_act_types: {},
                voc_act_call_types: {},
                voc_payment_status: {},
                voc_contract_type: {},
                voc_contract_wf: {},
                voc_contract_conclusion_basis: {},
                orgs_municipal: {},
                voc_amendment_types: {},
                voc_work_type_groups: {},
                voc_work_prj_status: {},
            })
```

Наборы опций пусты, значение имеют только имена справочников.

Они должны соответствовать полям `data`, заранее заполненным массивами словарных значений вида:

```
[
  {id: 1, label: "one"},
  {id: 2, label: "two", fake: 1},
  {id: 3, label: "three"},
  ...
  {id: 10, label: "ten"},
]
```

Каждый такой массив `add_vocabularies` превращает в объект вида

```
{
  items: [
    {id: 1, text: "one"},
         // id=2 отсутствует, поскольку было помечено fake=1...
    {id: 3, text: "three"},
    ...
    {id: 10, text: "ten"},
  ],
  1: "one",
  2: "two", // ...однако в словаре id=2 присутствует
  3: "three",
  ...
  10: "ten",
```

Таким образом, после обработки поля `voc_some_thing` для `list` и enum `w2ui` можно использовать `data.voc_some_thing.items`.

Кроме того, для каждого значения `id` данного справочника соответствующую надпись можно получить как `data.voc_some_thing [id]`.

Отметим обработку записи с логически истинным значением поля `fake` в вышеприведённом примере: она исключается из `list`, но присутствует в словаре. Таким образом помечаются строки справочников, которые:

- нельзя использовать для ввода новых данных (и поэтому надо фильтровать в `items`),
- но можно встретить в архивных записях (и, соответственно, необходимо расшифровывать по полному словарю).

открытие вкладки для созданного объекта (w2_confirm_open_tab)

Если popup-окно отправляет на сервер AJAX-запрос, создающий новую запись, часто требуется сразу открыть соответствующую страницу.

Однако непосредственно функцию open_tab здесь применять, к сожалению, нельзя: `window.open` по ходу обработки AJAX-ответа по умолчанию считается мошенническим и глушится браузером. Но если скомбинировать его с асинхронным w2confirm, эта проблема снимается:

```
data = await response ({type: 'planned_examinations', action: 'create'}, {data: v})
w2_close_popup_reload_grid ()
w2_confirm_open_tab ('Проверка создана. Открыть её страницу?', '/planned_examination/' + data.id)
```

отмена всех w2utils.lock() в окне (unlockAll)

перезагрузка UI блока, связанного с формой (w2form.reload_block)

Допустим, в системе реализован UI-блок:

- показывающий форму ввода,
- сгенерированную по шаблону,
- на одной из панелей layout'а,
  - которая определяется по индикатору ожидания.

Для таких форм часто требуется реализовать кнопку "Отмена", сбрасывающую все введённые данные и возвращающую панель с формой (включая видимость кнопок) в исходное состояние.

Данная задача решается однострочником вида:

```
$_DO.cancel_my_editing = function (e) {
 if (confirm ('Отменить несохранённые правки?')) w2_panel_form ().reload_block ()
}
```

который:

- устанавливает индикатор ожидания;
- вычисляет имя блока по содержимому панели (точнее, атрибуту `data-block-name`);
- инициирует его повторную загрузку.

поиск таблицы на экране (w2_first_grid)

Зачастую на странице UI присутствует не более одной таблицы (w2grid).

Функция `w2_first_grid` позволяет получить доступ к этой единственной таблице, найдя её имя в DOM-дереве страницы:

```
 function w2_first_grid () {
   return w2ui [$('.w2ui-grid').attr ('name')]
 }
```

Она используется в w2_close_popup_reload_grid, но может применяться и непосредственно в прикладном коде: например, для выяснения id текущей выделенной строки.

поиск текущей popup формы на экране (w2_popup_form)

Данная функция

```
function w2_popup_form () {
  return w2ui [$('.w2ui-popup .w2ui-form').attr ('name')]
}
```

находит на экране форму ввода внутри текущего открытого popup-окна. Предназначена для упрощения кода обработчиков таких блоков UI:

```
$_DO.update_unplanned_examinations_new_org = async function (e) {
 f = w2_popup_form ()
 var v = f.values ()
 if (v.ordernumber == null) die ('ordernumber', 'Необходимо указать номер приказа')
 if (v.orderdate   == null) die ('orderdate',   'Необходимо указать дату приказа')
 //...
 f.lock ()
 let data = await response ({type: 'planned_examinations', action: 'create'}, {data: v})
 w2_close_popup_reload_grid ()
 w2_confirm_open_tab ('Проверка создана. Открыть её страницу?', '/planned_examination/' + data.id)
}
```

поиск текущей формы ввода на странице (w2_panel_form)

Данная функция

```
 function w2_panel_form () {
   return w2ui [$('.w2ui-panel-content.w2ui-form').attr ('name')]
 }
```

находит на экране форму ввода внутри панели. Обычно такая форма одна — тогда неоднозначности не возникает. В более сложных случаях надо использовать объект `w2ui` и явное имя формы.

Предназначена для упрощения кода обработчиков таких блоков UI:

```
$_DO.update_planned_examinations = async function (e) {
 f = w2_panel_form ()
 var v = f.values ()
 if (v.ordernumber == null) die ('ordernumber', 'Необходимо указать номер приказа')
 if (v.orderdate   == null) die ('orderdate',   'Необходимо указать дату приказа')
 //...
 f.lock ()
 let data = await response ({type: 'planned_examinations', action: 'update'}, {data: v})
 reload_page ()
}
```

показ popup окна (w2uppop)

Модифицированный $.w2popup() с упрощённым набором параметров.

Допустим, в jQuery-объект view загружен HTML следующего вида:

```
  <span
    data-popup-width=600
    data-popup-height=220
    data-popup-title='Форма создания чего-то нового'
  >
    <div class="w2ui-reset w2ui-form" style="height:170px;margin:8px 0 0 0;">
      ...
      <div class="w2ui-buttons">
        <progress style="width:440px;display:none"/>
        <button class=w2ui-btn name=update>Сохранить</button>
      </div>
    </div>
  </span>
```

Тогда вызов

```
        $(view).w2uppop ({}, function () {

            $('#w2ui-popup .w2ui-form').w2reform ({
                name: 'some_form',
                fields: [
                  // ...
                ],
            })

        })
```

отобразит на экране popup с заданными (в HTML) размерами и заголовком и сразу запустит callback-функцию, которая создаст там форму и назначит обработчик `$_DO.update_...` на кнопку.

ВНИМАНИЕ! В w2ui есть досадная плавающая ошибка, из-за которой приходится **высоту для popup прописывать дважды**:

- в заголовке;
- в style внутреннего контейнера.

```
  <span
    data-popup-height=220
    ...
  >
    <div class="w2ui-reset w2ui-form" style="height:170px;margin:8px 0 0 0;">
```

Значения должны отличаться примерно на 50px, но это может зависеть от конкретного дизайна: необходимо подгонять по месту.

Если этого не делать, то UI может показаться работоспособным при первых тестах, однако непредсказуемо испортиться при многочисленных действиях с одним popup без перезагрузки экрана: внутренность формы будет выглядеть пустой. Похоже, при некоторых событиях w2ui зануляет высоту внутренних div в popup и не пересчитывает её.

Единственный известный скучный, но 100% надёжный способ избежать этого — явно указывать высоту внутреннего контейнера, как это описано выше.

получение набора нормализованных значений полей формы (w2form.values)

Выдаёт набор значений полей заданной формы, пригодный для отправки на сервер.

Результат имеет тот же тип, что у функции values из elu.js. Соответственно, может быть очищен и проверен при помощи `.actual ().validated ()`:

Пример:

```
 let v = w2_panel_form ().values ().actual ().validated ()
```

Отличия от содержимого w2form.record:

- присутствуют только поля, заявленные в w2form.fields (а не всё подряд, что было передано в .record при создании формы);
- строки очищены (trim) от концевых пробелов; строка без единого непробельного символа заменяется на `null`;
- даты приводятся к формату YYYY-MM-DD;
- значения list'ов передаются как id (а не объекты), enum - списки id;
- значения checkbox'ов - 0 или 1 (числа, не строки).
- у чисел убирается форматирование, дробная часть отделяется точкой;
- все числа, вводимые вручную (`int` и `float`), преобразуются в строки: в JSON они сериализуются с двойными кавычками.

экспорт в MS Excel (w2grid.saveAsXLS)

Экспорт содержимого w2grid с экрана в MS Excel, как в Eludia.pm, однако вообще без серверного программирования:

```
  $(w2ui ['main'].el ('main')).w2regrid ({
    name: 'lift_worksGrid',
    toolbar: {
      items: [
        {type: 'button', id: 'printButton', caption: 'MS Excel', onClick: function (e) {this.owner.saveAsXLS ()}},
  ...
```

Реализация в основном аналогична $.saveAsXLS из elu.js с тем отличием, что данные берутся не из HTML-таблицы, а из объекта w2grid. В частности, если таблица динамическая и отображена часть данных, то `w2grid.saveAsXLS` самостоятельно инициирует все недостающие запросы.

Переходник для SlickGrid и jQueryUI

draw_form

Заполнение HTML-шаблона (см. to_fill в elu.js) с инициализацией режима чтения/редактирования.

Пример:

```
$_DRAW.employee = async function (data) {
    $('title').text (data.label)

    let $form = await draw_form ('employee', data)

    return $('main').html ($form)
}
```

См. также: $.on_change.

Режим чтения/редактирования

Если шаблон содержит кнопки (BUTTON) с именами `edit` и `cancel`, то:

- при первом показе кнопка `cancel` скрыта, а у всех полей ввода установлен атрибут `disabled`;
- при нажатии кнопки `edit`:

    - сама она скрывается;
    - `cancel` показывается;
    - `disabled` убирается;
- при нажатии кнопки `cancel` область экрана повторно заполняется исходными данными (текущие значения полей ввода теряются) и принимает первоначальный вид (с атрибутами `disabled`).

Если хотя бы одна из кнопок `edit`/`cancel` отсутствует, описанные действия не производятся.

$.on_change

Расширение jQuery, которое для заданного поля ввода (INPUT, TEXTAREA, SELECT):

- назначает обработчик события `change`,
    - на вход которому, в отличие от стандартного $.change (), подаётся не событие, а значение поля;
- тут же инициирует его.

Предназначено для инициализации полей-переключателей (чаще всего это `checkbox` и `radio`), управляющих видимостью логически зависимых областей экрана.

В примере ниже сразу после отрисовки карточки учётной записи системы устанавливается связь: элемент с `id=td_roles` (клетка с полем выбора ролей) виден тогда и только тогда, когда `login` непуст.

```
let $form = await draw_form ('user', data)
$('input[name=login]', $form).on_change (v =>
  $('#td_roles', $form).css ({display: v == null ? 'none' : 'table-cell'})
)
```

Поскольку `on_change` не только устанавливает обработчик, но и вызывает его, условие сразу проверяется для исходных данных формы, до каких-либо действий пользователя.

$.valid_data

Обёртка над функцией values из elu.js: она тоже возвращает объект со значениями полей в заданной области, но только если они удовлетворяют правилам валидации, прописанным в атрибутах полей ввода (`required`, `min`, `max` и т. п.). В противном случае генерируется исключение, которое приводит к показу `alert` с ошибкой и фокусировке на проблемном поле.

```
 let $form = get_popup ()
 let  data = $form.valid_data ()
```

draw_popup

Комбинация из

- заполнения HTML шаблона (см. to_fill в elu.js) и
- отображения popup-окна средствами jQueryUI dialog

```
 /* my_block_name.html:
    <span width=800 height=600 title="Окошечко">
      ...
      <button name=update data-hotkey=Ctrl-Enter>Применить</button>
    </span>
 */
 let $view = await draw_popup ('my_block_name', data) // по-простому
// let $view = await draw_popup ('my_block_name', data, popup_options)
// если вдруг нужны спецопции
```

В отличие к jQueryUI dialog:

- устанавливает по умолчанию (что может быть явно переопределено в `popup_options`):
    - `modal=true`;
    - `dialogClass` — имя блока (в данном примере `my_block_name`);
- помимо атрибута `title` для заголовка, берёт из предоставленного контейнера:
    - атрибуты
        - `width`
        - `height`
        - `noresize` (при котором ставит `resizable: false`)
    - элементы `BUTTON` верхнего уровня — они убираются из заполненного HTML-шаблона и преобразуются в соответствующую опцию `buttons`
- по событию `dialogclose`:
    - блокирует обработку события, что предотвращает перехват нажатия `Esc` на уровне страницы;
    - не только скрывает popup, но убирает из DOM связанные с ним элементы, что позволяет многократно вызывать `draw_popup` на одной странице с одинаковыми параметрами без накопления ошибок;

draw_table

Создание SlickGrid с некоторыми расширениями.

В отличие от конструктора SlickGrid, здесь всего один аргумент: тот, который в оригинале называется `options`. Переменные `columns` и `data` вставлены туда как компоненты, а `container`, наоборот, вынесен, поскольку `draw_table` реализована как расширение jQuery.

```
    let grid = $("#grid_my_type").draw_table ({
        editable: data._can.edit,
        // ...прочие скалярные опции

        columns: [/* см. ниже */].filter (not_off),

//      data: [{id: 1, label: 'Один'}]
          // это для таблиц с фиксированными данными
```

```
        src: ['my_type', {search: [{field: 'is_deleted', operator: 'in', value: [0]}]}],
          // это для таблиц с подкачкой
        searchInputs: $(".toolbar_filters *").toArray (),
          // ссылка на поля поиска вне самой таблицы
        onRecordDblClick: (r) => open_tab ('/my_type/' + r.uuid),
        // ...прочие события

    })
// ... и далее этот объект можно получить так:
    let grid = $("#grid_my_type").data ('grid')
```

Источник данных

Статический массив

Когда требуется отобразить таблицу для фиксированного набора строк -- соответствующий массив следует указать в качестве опции `data`.

Если при этом на момент вызова `draw_table` вычисленная высота контейнера равна нулю, то автоматически включается опция `autoHeight` и SlickGrid устанавливает высоту ровно такой, чтобы отобразить заданный набор строк.

Динамическая подкачка (AJAX)

В API SlickGrid динамическая передача данных частями ограниченного объёма отсутствует. А типовой пример на эту тему не только немасштабируем (поскольку привязан к строго определённому сайту-источнику), но в настоящее время даже неработоспособен (по той же причине).

Поэтому в elu_slick динамический источник данных реализован практически с нуля.

RESTful API позаимствован у w2grid -- соответственно, серверные части для клиентов, использующих draw_table и w2grid, взаимозаменяемы. Также по аналогии с w2ui в API добавлены опции `url` и `postData`. Однако использовать в прикладном коде рекомендуется не их а описатель источника данных `src`.

В общем случае значением `src` является массив из двух элементов: из первого формируется `url` (точнее, первый параметр query), а второй совпадает со значением `postData` по умолчанию. Пример:

```
 src: [
  'users.log', // -> /?type=users&part=log
  {search: [{field: "_uuid", value: $_REQUEST.id, operator: "is"}]}
 ],
```

Если значение первого параметра содержит точку (как в примере выше), она считается разделителем `type` и `part`, иначе эта стока принимается за `type` целиком. Параметры `id` и `action` всегда пусты. Ссылки на родительские объекты надо включать во второй параметр -- `postData` (снова см. пример).

При отправке AJAX-запросов у `postData` дополнительно устанавливаются компоненты:

- `limit`: размер порции строк, по умолчанию `50`, можно установить явно;
- `offset`: начало требуемой порции -- его grid вычисляет самостоятельно.

В простейшем (но распространённом) случае реестра с пустой `part` и без обязательных фильтров в качестве src можно указать только первый элемент: `type`. Часто он совпадает с именем блока текущей страницы:

```
src: $_REQUEST.type,
```

Всё остальное будет вычислено автоматически.

Определения столбцов

В основном элементы `columns` имеют такой же формат, как в SlickGrid. Расширения перечислены ниже.

Словарь данных: voc

Если значения поля таблицы -- коды в справочнике, предварительно обработанном add_vocabularies, а показывать требуется соответствующие поля `label` из словаря, то вместо вместо написания `formatter'a` по месту можно указать сам словарь:

```
{
  field: "id_voc_org",
  name: "Организация",
  width: 100,
  sortable: true,
  voc: data.voc_orgs,
}
```

Фильтр по столбцу: filter

При вызове `draw_grid` для каждого столбца можно определить локальный фильтр: тогда в соответствующей клетке первой строки таблицы отобразится поле поиска, как в MS Excel. Это имеет смысл для таблиц с динамической подкачкой: установленные значения фильтров включаются в `postData`, после чего набор данных перезагружается AJAX-запросом.

elu_slick реализует три стандартных типа фильтров:

- `text` (поле поиска в верхней клетке),
- `dates` (popup выбора вилки дат) и
- `checkboxes` (popup с множественным выбором в таблице словарных записей)

```
          {
              field: "dt",
              name: "Дата",
              width: 100,
              sortable: true,
              filter: {type: 'dates', title: '[Все даты]',
                dt_from: 'YYYY-MM-01', // предлагается начало месяца
                dt_to:   'YYYY-MM-DD', // предлагается текущая дата
```

```
            },
        },
        {
            field: "subject",
            name: "Тема",
            width: 100,
            sortable: true,
            filter: {type: 'text', title: '[Все темы]'},
        },
        {
            field: "uuid_organization_customer",
            name: "Заказчик",
            width: 100,
            voc: data.vw_contract_customers,
            filter: {
                type: 'checkboxes',
                title: 'Заказчик',
//                  items: data.vw_contract_customers.items, // можно не указывать -- используется voc
            },
        },
```

Технически это блоки elu.js с именами `_grid_filter_dates`, `_grid_filter_text` и `_grid_filter_checkboxes` соответственно.

На уровне приложения можно как переопределить их логику, так и реализовать новые типы фильтров: достаточно использовать префикс `_grid_filter_`.

Поля выбора

SlickGrid можно использовать в качестве поля единичного или множественного выбора. В этом разделе показано, как именно.

radio

Для выбора ровно одной строки всё обстоит весьма просто:

```
{
  field: 'uuid',
  name: "",
  maxWidth: 20,
  formatter: (r, c, v) => `<input type=radio name=uuid value=${v} required>`
},
```

Здесь используется стандартная для SlickGrid возможность задать HTML-шаблон отображения клетки. В результате первый столбец будет содержать радио-кнопки, значения которых можно получать стандартным образом: например, функцией $.valid_data.

checkbox

С галочками аналогичный фокус может пройти только если все строки таблицы видны одновременно, то есть она не допускает прокрутки. Скрывшиеся за горизонтом клетки SlickGrid оперативно удаляет из DOM — вместе со значениями полей, если они там есть.

Для полноценного множественного выбора необходимо, чтобы проект были добавлены `slick.rowselectionmodel.js` и `slick.checkboxselectcolumn.js` — тогда можно определить столбец

```
{
   //  hideInColumnTitleRow: true, — обычно имеет смысл
  class: Slick.CheckboxSelectColumn,
   // `class` — опция, добавленная в elu_click: класс объекта, который станет описанием столбца
   // Slick.CheckboxSelectColumn — определен в `slick.checkboxselectcolumn.js`
   //   Slick.RowSelectionModel явно создавать не надо — draw_table делает это самостоятельно
},
```

и далее устанавливать выбранное множество `ids` следующим образом

```
  let idx  = {}; for (let id of ids) idx [id] = 1
  let rows = []; for (let i = 0; i < data.length; i ++) if (idx [data [i].id]) rows.push (i)
  grid.setSelectedRows (rows)
```

а получать его — так:

```
  let grid = $("#grid_options").data ('grid')
  let ids = grid.getSelectedRows ().map (i => grid.getDataItem (i).id)
```

Поля редактирования

Для таблиц с установленной опцией `editable` нижеописанные поля можно применять без подключения каких-либо дополнений.

Slick.Editors.Input

Отображает во всю клетку (100% высоты и ширины) стандартный HTML-элемент INPUT с набором атрибутов, заданным опцией `input`. Пример:

```
        {
            field: "duration_days",
            name: "Раб. дней",
            width: 1,
            editor: Slick.Editors.Input,
            input: {
                type: 'number',
                min: 0,
                max: 256,
                step: 1,
            }
        },
```

Slick.Editors.Select

Отображает во всю клетку (100% высоты и ширины) стандартный HTML-элемент SELECT с набором строк, заданным опцией `voc` (той же самой, что используется для отображения и фильтрации -- см. выше). Пример:

```
            {
                field: "id_voc_voc_inspection_form",
                name: "Форма",
                width: 1,
                editor: Slick.Editors.Select,
                voc: data.voc_inspection_forms,
                empty: '[не назначено]',
            },
```

События

onCellChange

Если среди опций определён обработчик `onCellChange`, он оборачивается таким образом, что при возникновении ошибки редактирование прерывается с восстановлением исходного значения в поле.

onRecordDblClick

К стандартному набору событий добавлено `onRecordDblClick`:

```
onRecordDblClick: (r) => open_tab ('/my_type/' + r.uuid),
```

В отличие от обработчиков событий SlickGrid, здесь на вход подаётся элемент данных, соответствующий отдельной записи -- обычно это то, что требуется в данном случае. При необходимости можно воспользоваться стандартным `onDblClick`. Отметим, что для таблиц с опцией `enableCellNavigation` (кстати `draw_grid` устанавливает её по умолчанию, позволяя явно прописать `false`) событие `dblClick` дополнительно вызывается при нажатии клавиши `Enter`. Что позволяет использовать таблицу для клавиатурной навигации в стиле Norton Commander.

Прочее

Экспорт в Excel

Для заданной таблицы на клиенте можно сформировать файл, пригодный для просмотра в MS Excel:

```
let $this = $("#grid_organizations")
let grid = $this.data ('grid')
$('progress').attr ({value: 0, max: grid.getData ().length})
await grid.saveAsXLS ('Юридические лица.xls', value => $('progress').attr ({value}))
close_popup ()
```

На самом деле формируется строка HTML, которую предлагается сохранить с расширением ".xls", но MS Excel показывает такие файлы как электронные таблицы, хотя и выдаёт сообщение о несоответствии формата.

Если таблица привязана к динамическому источнику данных, её содержимое будет прочитано постранично, полностью. Поскольку результат накапливается в памяти, при большом количестве строк это может вызвать переполнение на стороне клиента.

Серверная библиотека Dia.js

Dia.js — это библиотека для разработки на базе node.js серверного ПО, использующего реляционные СУБД.

Сфера применения — Web-сервисы разного рода: в частности, серверные части информационных систем с Web-интерфейсами, мобильными клиентами и т. п. Впрочем, протокола HTTP не является принципиальным. Dia.js — основа для систем массового обслуживания запросов произвольных форматов.

Структура директорий

Общие слова

В рамках проектов на Dia.pm принята фиксированная структура директорий проекта и именования функций. Имея перед глазами конкретный запрос, разработчик может сразу открыть нужный файл и найти там функцию, обрабатывающую эти данные, не теряя времени на вопросы из серии: *"Чем же оно это сказало?.."*

Конкретика

В проекте типичного приложения с Web-интерфейсом на верхнем уровне имеется две директории:

- `/front`, где хранится то, что должно загружаться на клиент и исполняться в браузере;
- `/back` — программный код, исполняемый на сервере.

Параллельно `/front` могут находится несколько аналогичных директорий (отдельный Web UI для публичной части, исходники мобильного приложения и т. п.), но речь сейчас не о них.

Директория `/back` обычно одна и ниже описано её содержимое.

Итак, `/back`

- делится на две части:
  - рудиментарную `/back/conf`, о которой рассказано в разделе Конфигурация;
  - и `/back/lib` — описание логики приложения;
- кроме того, содержит `package.json` — соответственно, в ней предусмотрен запуск `npm i`, приводящий к загрузке `node_modules`, находящихся вне git.

Последняя, в свою очередь, содержит:

- главный пусковой файл `/back/lib/index.js`;
- директорию описания модели данных `/back/lib/Model`;
  - каждый файл в этой директории соответствует по имени таблице ики VIEW в базе данных;
  - у приложений, работающих с более, чем одной БД — будет соответствующее число директорий моделей;
- и директорию с текстами обработчиков запросов `/back/lib/Content`
  - файлы в самой этой директории — модули для работы с конкретными типами данных (что соответствует понятию *ресурсов* в терминах REST);
  - поддиректория `back/lib/Content/Handler` содержит программный код обработчиков, используемый для многих типов, однако специфичный для данного приложения, а не Dia.js в целом.

Сама Dia.js располагается не в `node_modules`, а в `back/lib/Ext` на правах подмодуля git.

Конфигурация

Общие слова

Прежде, чем комментировать конкретные файлы, отметим важный принцип:

`значения параметров конфигурации приложения не могут входить в состав программного кода.`

Строки подключения к БД, адреса почтовых серверов, сторонних Web-сервисов, ключи доступа к ним, значения допустимых времён ожидания и все прочие строки и числа, которые могут изменяться от среды к среде, могут храниться в каких угодно файлах и прочих источниках данных, только не в тексте программы, отслеживаемом версионным контролем.

Несмотря на довольно очевидную мотивировку данного положения, многие разработчики его прямо игнорируют. В частности, широко известна недобрая традиция писать пароль к БД и одновременно вызовы подключаемых модулей в нечто под названием `Config.php`.

Расположение файла

В нашем типовом приложении схема хранения параметров среды, хотя и незатейлива, но всё же не до такой степени.

Предполагается, что значения параметров пишутся в формате JSON в файл `elud.json`, расположенный в директории back/conf. В коде приложения эта директория присутствует, но там лежит не сам `elud.json`, а пример, с которого его можно скопировать: `elud.json.orig`. Сам же `elud.json` прописан в .gitignore.

Путь к файлу `back/conf/elud.json` в рассматриваемой версии прописан жёстко, однако это вовсе не обязательно. Типовое усовершенствование: читать этот путь из переменной окружения, которую должен устанавливать пусковой скрипт.

Чтение конфигурации

Для загрузки конфигурации из этого файла реализован js-класс Config, используемый в index.js.

Помимо очевидных чтения и разбора JSON-файла, здесь сразу создаётся pool соединений с БД, который далее будет повсеместно использоваться именно как компонент, получаемый из объекта конфигурации.

Наше приложение использует только одну БД, этого может оказаться достаточно для многих приложений. В методах обработки запросов связь с ней доступна через переменную `this.db`. Так принято, это удобно, но при желании можно переименовать.

Как правило, при наличии множества подключений одна БД по смыслу всё-таки является главной, "родной". Однако в API это никак не выделяется. Компонента `this.db` может и отсутствовать.

Кроме того, в `Config` можно добавить чтение параметров и, соответственно, создание pool'ов для подключения ещё к произвольному числу как

реляционных БД, так и источников данных иной природы (LDAP, noSQL) и прочих ресурсов. Они также окажутся видны в качестве компонент `this`.

Модель БД

Dia-приложение, подключённое к реляционной БД, не просто способно направлять туда SQL-запросы и получать результаты. Оно знает:

- какие ему требуются таблицы;
- какие у них должны быть столбцы;
  - в том числе, какие из них являются ссылками и куда именно;
- какое требуется наполнение (для фиксированных справочников);
- какие индексы;
- какие триггеры

и умеет:

- проверять наличие этого всего в предоставленной БД;
- при отсутствии -- создавать или преобразовывать в сторону расширения (увеличивать размерность полей, снимать NOT NULL).

И далее на основании этих же знаний о структуре данных API позволяет использовать для запросов простые компактные JSON-объекты вместо громоздких SQL-конструкций.

Модель как объект

Для каждой используемой базы данных приложение может определять собственную модель:

- класс, наследуемый от Dia.db.Model;
- поддиректорию `/back/lib` с классами-описаниями таблиц (обычно /back/lib/Model).

На этапе чтения конфигурации приложение создаёт экземпляр модели:

```
let model = new (require ('./Model.js')) ({path: './Model'})
```

и передаёт его как параметр вновь создаваемому ресурсу: pool'у соединений с БД:

```
this.pools = {
  db: Dia.DB.Pool (this.db, model),
  //...
}
```

Как правило, объект конфигурации реализует также метод `init`, обеспечивающий первичную миграцию данных и вызываемый из index.js до начала приёма запросов.

Далее при исполнении каждого метода любого модуля в качестве переменной `this.db` будет доступно соединение с БД, использующее данную модель. Прикладной код будет видеть:

- саму модель как `this.db.model`;
- справочник её таблиц -- как `this.db.model.tables`;
- определение столбца `c` таблицы `t` -- как `this.db.model.tables [t].columns [c]`;
- гарантированное наполнение таблицы `t` -- как `this.db.model.tables [t].data`.

Элементы описания таблиц

Общий формат

Описание таблицы представляет собой js-модуль, экспортирующий единственный объект:

```
module.exports = {
 ...
}
```

Физическое имя таблицы в БД совпадает с именем файла (без расширения `.js`, разумеется), так что в данном случае описывается таблица `roles`.

Комментарий к таблице

Все таблицы и каждое их поле в модели данных прокомментированы. Комментарий к таблице задаётся компонентой `label`:

```
  label: 'Роли',
```

Столбцы

Поля таблицы описываются в элементе `columns`, где ключами являются физические имена полей:

```
   columns: {
      //...
   },
```

Для описания опций хранения полей применяется микроязык, описанный ниже.

Простейшие скалярные поля

Минимальное описание поля сводится к указанию его типа и комментария:

```
ttl:  'int // Время на исполнение, с',
```

В качестве имён типов поддерживаются как родные для СУБД, так и транслируемые для совместимости:

- `bigint`
- `checkbox` (как `tinyint`)
- `datetime`
- `mediumint`

- money
- number
- smallint
- string (как varchar)
- text
- tinyint

Размерность

Размерность поля указывается в квадратных скобках после имени типа: одним натуральным числом (длина поля):

```
label: 'string [30] // ФИО',
```

или двумя (с дробной частью):

```
salary: 'money [8,2] // Оклад',
```

DEFAULT / NOT NULL

Значение по умолчанию указывается после знака =. Оно может быть как константой, так и функцией

```
is_deleted : 'int=0                    // 1, если удалена',
uuid       : 'uuid=uuid_generate_v4() // Ключ',
```

Для таких полей устанавливается опция NOT NULL. Все остальные допускают пустые значения.

Поля-ссылки

У поля, значение которого подразумевается значением первичного ключа в таблице той же модели, вместо имени типа ставится имя целевой таблицы в скобках:

```
id_user        : "(users)                                          // У кого на рассмотрении",
id_user_author : "(users)=current_setting('tasks.id_user')::uuid // Автор",
```

Объявить поле-ссылку на таблицу, первичный ключ которой содержит более одного поля -- невозможно. В корректно спроектированной модели такой потребности и не возникает, поскольку таблицы с составными ключами сами по себе не должны представлять ничего, кроме множественных обратных ссылок.

Регулярные выражения для проверки

С каждым полем можно ассоциировать регулярное выражение.

```
label: 'string [30] /^[А-ЯЁ][А-ЯЁа-яё \\-]+[а-яё]$/ // ФИО',
```

По умолчанию оно не будет как-либо использоваться, а лишь станет доступно в качестве компоненты PATTERN объекта, представляющего данное поле в модели. Чтобы реализовать соответствующую проверку, необходимо либо описать соответствующий триггер (см. ниже), либо использовать PATTERN на уровне модуля.

Индексы

Первичный ключ

Для каждой таблицы должен быть определён первичный ключ. Он задаётся опцией pk:

- если ключ простой, то как имя поля:

```
pk: '_bizarre_key__f131D',
```

- если составной -- как массив имён:

```
pk : ['id_user', 'id_user_ref'],
```

В норме указание первичного ключа должно быть частью общих соглашений как минимум уровня всей модели и реализовываться в методе on_before_parse_table_columns соответствующего объекта, наследующего Model. Например, следующий фрагмент кода

```
    on_before_parse_table_columns (table) {
        let cols = table.columns

        if (!table.pk) {
          if (cols.id) {
            table.pk = 'id'
          }
          else {
            cols [table.pk = 'uuid'] = 'uuid=uuid_generate_v4()'
          }
        }

    }
```

гарантирует, что за исключением отдельно описанных ключей:

- у таблиц с полем id первичным ключом будет поле id (имеются в виду фиксированные справочники: статусов, видов типов классов и т. п.);
- у всех прочих первичным ключом будет поле uuid;
  - причём там, где оно не заявлено в модуле таблицы, общая Model добавит его автоматически.

Подробнее о составных первичных ключах

Отметим, что указание более одного ключевого поля при описании таблицы налагает жёсткое ограничение: в других таблицах невозможно объявлять соответствующие поля-ссылки (см. выше). Соответственно, this.db.query не может автоматически вычислять условия JOIN для таких таблиц -- хотя при их ручном указании проблем не возникает.

Составные ключи должны использоваться только в чисто связных таблицах "многие-ко-многим", обычно их относительно немного. Поэтому для большинства таблиц опция `pk` является строкой.

Однако для единообразия модель автоматически добавляет в описание компоненту `p_k`, значение которой -- всегда массив (js Array) имён ключевых полей. Соответственно, для произвольной `table` из `this.db.model.tables`:

- проверка на простой первичный ключ:

```
if (table.p_k.length == 1) ...
```

- проверка вхождения столбца с именем `col_name` в первичный ключ:

```
if (table.p_k.includes (col_name)) ...
```

Прочие индексы

Остальные (вторичные) ключи описываются в разделе `keys`:

```
keys : {
 label   : 'label',
 id_task : 'id_task,is_author',
 id_user : 'UNIQUE (id_user,id_voc_user_option)',
},
```

Данные

Базы данных многих приложений содержат таблицы, по своей сути являющиеся перечислимыми типами, неразрывно связанными с бизнес-логикой. Типичные примеры: виды документов, удостоверяющих личность, категории водительских прав, роли пользователей, статусы документов.

Разумеется, любой справочник может потребовать настройки и расширения, однако в логике каждой версии реально работающей системы какие-то коды обязательно прибиты гвоздями намертво. Иначе ничего не будет работать.

Фиксированное наполнение

Гарантированное содержимое любой таблицы можно задать в модели, наряду с её структурой:

```
 data: [
   {id: 1, name: 'admin', label: 'Администратор'},
   {id: 2, name: 'user',  label: 'Пользователь'},
 ],
```

Более того, поскольку модель всегда загружена в память, их можно и извлекать оттуда, не обращаясь к таблице.

Отметим, что SQL, генерируемый из `data`, затрагивает только явно упомянутые там поля. То есть после исполнения скрипта миграции при ранее существующих данных для рассмотренного случая у записи с `id=1` значение поля `label` обязательно будет установлено 'Администратор'. Но если при этом в таблице есть ещё поле, скажем, `label_full` и там что-то лежит с прошлого запуска системы — оно не изменится. Правда, и не будет извлечено методом this.db.add_vocabularies.

Начальное наполнение

Некоторые таблицы должны содержать известные строки только непосредственно после создания, а потом из записи могут редактироваться и исходные данные вовсе не должны всплывать при каждой перезагрузке.

Для этого предусмотрена опция `init_data`:

```
 init_data: [
   {id: 1, login: 'admin', label: 'Условный админ на первый вход'},
 ],
```

Загрузка из внешнего источника

Значением компонент `data` и `init_data` может быть не только список записей, но и (асинхронная) функция, генерирующая его.

Это удобно, в частности, для выноса тел справочников в файлы, включаемые в исходные тексты приложения, но имеющие синтаксис, отличный от javaScript: например, табулированный текст:

```
    data: function () {

        return this.model.read_data_lines (this.name).map (s => {
            let    [id, id_voc_contract_type, name, label] = s.split ("\t")
            return {id, id_voc_contract_type, name, label}
        })
    },
```

Триггеры

Набор триггеров в описании таблицы имеет предсказуемое название и представляет собой js-объект, где ключами являются специально составленные имена (фаза_событие1_..._событиеN), а значениями -- исходные тексты соответствующих блоков:

```
 triggers : {
   before_insert_update : function () {
     return this.model.trg_check_column_values (this) + 'RETURN NEW;'
   },
   // ...
 },
```

Данный пример взят из типового приложения, функция-генератор кода `trg_check_column_values` определена на уровне модели в целом.

Если текст триггера (за вычетом пробельных символов) начинается со слова `DECLARE`, он используется как есть.

В противном случае текст обрамляется скобками `BEGIN ... END;`.

Описание VIEWs

SQL VIEWs -- объекты, внешне весьма сходные с таблицами, однако лишённые физических параметров хранения и базирующиеся на запросах типа SELECT.

Dia.DB.Model поддерживает описания VIEWs в виде объектов, содержащих точно такие же, как у таблиц, компоненты `columns` и `pk`, и компоненту `sql` с исходным текстом запроса. Пример:

```
    label: 'Активные пользователи + роли',
    columns: {
        label: "text // Имя",
        role:  "text // Роль",
    },
    sql: `
        SELECT
            users.uuid
            , users.label
            , roles.name AS role
        FROM
            users
            INNER JOIN roles ON users.id_role = roles.id
        WHERE
            users.is_deleted = 0
    `,
```

Набор полей в выражении `SELECT` должен соответствовать содержимому `columns` -- с учётом его обработки на уровне `Model`. В приведённом примере на этом этапе добавляется и объявляется первичным ключом поле `uuid`.

Миграция данных

Как правило, на старте приложения имеет смысл получить либо гарантии того, что предоставленная БД имеет необходимую структуру, либо, по крайней мере, ошибку, свидетельствующую о невозможности этого добиться -- чтобы не стартовать в заведомо неработоспособном состоянии.

В норме в Dia-приложении это должно выполняться в методе `init` объекта конфигурации, вызываемом из index.js до начала приёма запросов.

Простой вариант такого метода включает два вызова:

```
async init () {
 let db = this.pools.db
 await db.load_schema ()  // чтение физической схемы БД
 await db.update_model () // обновление до требуемого состояния
}
```

Иногда монолитный метод `update_model ()` оказывается неприменимым в силу того, что по ходу миграции требуются некоторые дополнительные действия. Тогда можно разделить процесс на 3 шага:

- генерация скрипта методом `db.gen_sql_patch ()`;
- его преобразование;
- запуск методом `db.run ()`.

Скрипт на выходе `db.gen_sql_patch ()` (и, соответственно, на входе `db.run ()`) имеет вид списка объектов с компонентами `sql` и `params`:

```
[
 {sql: '...', params: [...]},
 //...
 {sql: '...', params: [...]},
]
```

Что с ним будет производиться до (или даже вместо) запуска посредством `db.run ()` -- на ответственности разработчика конкретного приложения.

Разумеется, нет никаких препятствий тому, чтобы `db.run ()` был запущен необходимое число раз, в том числе с параметрами, не имеющими отношения к `db.gen_sql_patch ()`.

События миграции

В Dia.js делается всё возможное, чтобы разработчик не писал `CREATE` и `ALTER` вручную, а лишь задавал параметры схемы данных, предоставляя автомату генерировать необходимый DDL. Такой подход противопоставляется отслеживанию явных преобразований (Liquibase, Flyway и т. п.) как управление с обратной связью -- программному управлению.

Тем не менее, по ходу развития логики приложения неизбежно возникают моменты, когда требуется мигрировать накопленные данные при помощи алгоритма, который в принципе невозможно вывести из одних только типов и размерностей новых полей.

Dia.db.model позволяет реализовать и такие действия при помощи элементов описания таблиц -- они имеют вид отрезков скриптов такого же вида, как `db.gen_sql_patch ()` (и вставляемые в список на его выходе непосредственно).

on_after_add_column

Это раздел описания таблицы, одного уровня с `columns`, содержащий мапинг имён полей на отрезки скриптов, которые должны запускаться после добавления соответствующих полей, если таблица в целом на момент загрузки схемы уже существовала.

Поясним сказанное на примере из истории реального приложения: системы учёта задач.

Каждая реплика в переписке по задаче может иметь единственною иллюстрацию: файл, путь которого жёстко связан с датой и номером самой реплики. До версии 428cc46 эти пути вычислялись при каждом использовании, далее было принято решение сделать данный реквизит хранимым и вычислять лишь единожды. Все экземпляры БД приложения (в том числе боевой) на тот момент содержали данные -- соответственно, возникла задача вычисления путей файлов-иллюстраций для существующих реплик.

Решением задачи явилось добавление к описанию таблицы реплик, помимо самого нового поля `path` в раздел `columns`, ещё верхнеуровневого раздела `on_after_add_column` с одноимённой компонентой `path`:

```
    on_after_add_column: {
```

```
        path: [
            {sql: `UPDATE task_notes SET
              path = TO_CHAR(ts, 'YYYY/MM/DD/') || uuid || '.' || COALESCE (ext, 'png')
              WHERE is_illustrated = 1`
            , params: []}
        ],

    }
```

on_after_add_table

Это вехнеуровневая опция описания таблицы, значением которой является отрезок скрипта, запускаемый после `CREATE TABLE`.

Характерный сценарий использования: добавление нового отношения многие-ко-многим для ранее накопленного массива данных. Снова приведём пример из истории системы учёта задач.

До версии 95d28f6 задачи не имели ссылок друг на друга. Далее было принято решение добавить горизонтальные связи на основании содержимого: по URL, входящим в тексты реплик (благо такие URL оканчиваются на UUID, являющиеся значениями первичного ключа в таблице задач).

Необходимое наполнение на каждой среде (тестовых и боевой) было обеспечено следующим фрагментом определения вновь создаваемой таблицы:

```
on_after_add_table: {
  sql: `

    INSERT INTO task_tasks (
      id_task,
      id_task_to
    )
    SELECT t.* FROM (
      SELECT DISTINCT
        id_task
        , RIGHT((REGEXP_MATCHES (label || body,
          'tasks/[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}'
          , 'g'))[1], 36)::uuid id_task_to
      FROM
        task_notes
    ) t INNER JOIN tasks ON t.id_task = tasks.uuid
  `, params: []
}
```

on_before_create_index

Это раздел описания таблицы, одного уровня с `keys`, содержащий мапинг имён ключей на отрезки скриптов, которые должны запускаться перед созданием соответствующих индексов, если таблица в целом на момент загрузки схемы уже существовала.

Данный механизм предусмотрен для случая, когда после некоторого периода эксплуатации выясняется, что записи таблицы должны быть уникальными по некоторому набору полей — но там уже накоплены дубли, противоречащие этому условию. В этой ситуации добавление ключа с опцией CREATE UNIQUE INDEX приведёт к ошибке на старте приложения. Но если перед ним добавить скрипт зачистки — то проблемы можно избежать. Пример:

```
keys : {
  uuid_insp_order: 'UNIQUE (uuid_insp_order, uuid_voc_norm_act)',
},
on_before_create_index: {
  uuid_insp_order: [

    {sql: 'DELETE FROM insp_order_norm_acts WHERE is_deleted=1', params: []},

    {sql: `

       DELETE FROM insp_order_norm_acts WHERE uuid IN (
         SELECT dups.uuid
         FROM insp_order_norm_acts dups
         LEFT JOIN insp_order_norm_acts root on root.uuid <> dups.uuid
           AND root.uuid_insp_order = dups.uuid_insp_order
           AND root.uuid_voc_norm_act = dups.uuid_voc_norm_act
         WHERE
           root.uuid < dups.uuid
       )

    `, params: []},

  ]
}
```

on_before_recreate_table

Это верхнеуровневая опция описания таблицы, значением которой является функция-генератор скрипта, запускаемого перед пересозданием таблицы. В качестве единственного параметра функция получает старое описание таблицы, извлечённое из БД:

```
    on_before_recreate_table: (table) => {
        if (table.existing.columns.is_on) return {
      sql    : 'DELETE FROM user_users WHERE is_on=?',
          params : [0],
        }
    }
```

Само по себе пересоздание происходит в том случае, когда у таблицы меняется первичный ключ. Это, в частности, может быть:

- переход к типу с более широкой областью значений (например, от `int` к `uuid`);
- переход от простого ключа с составному и наоборот.

Если новый первичный ключ является простым (состоит из единственного поля), то помимо данных самой таблицы производится миграция ссылок на неё: все существующие поля-ссылки пересоздаются с новыми типами и заполняются соответствующими вычисленными значениями.

Как перезапустить?

Ошибки по ходу разработки -- дело житейское, так что для любого фрагмента процедурного кода крайне желательна возможность многократного повторного исполнения.

Механизм, запускающий скрипты, привязанные к `on_after_add_table` и `on_after_add_table`, предполагает единственность запуска.

Однако, поскольку решение о запуске/пропуске для большинства описанных событий принимается на основании легко проверяемого свойства БД -- наличия таблицы или поля в ней (а не, например, контрольной суммы, возможно, хранящейся вне самой БД) -- то и повлиять на него очень просто: соответствующим `DROP`.

Что же касается достаточно редкого события `on_before_recreate_table`, то для его отладки необходимо вовремя запастись эталонной копией данных и восстанавливать перед запуском миграции до тех пор, пока не будет достигнут желаемый результат.

Генератор SQL

`this.db.query` — это функция, генерирующая SQL-запрос и соответствующий набор параметров из js-структуры данных, напоминающей запросы NoSQL-БД вроде MongoDB, CouchDB и им подобных.

Использование

`this.db.query` предназначена для использования в коде приложения не напрямую, а для промежуточной подготовки параметров в высокоуровневых функциях:

- this.db.fold — проход по выборке (this.db.select_loop)
- this.db.get — получение отдельной записи (this.db.select_hash)
- this.db.list — получение списка (this.db.select_all)
  - а также this.db.add и this.db.add_all_cnt

Хотя изредка имеет смысл вызвать её непосредственно и воспользоваться результатом вида:

```
{sql: 'SELECT ...', params: [...]}
```

Общий вид запроса

Запрос представляет собой непустой список *частей*, то есть объектов, каждый из которых содержит требования к одной из таблиц в запросе:

- что это за таблица;
- какие нужны поля;
- как её связать с остальными частями;
- какие наложить фильтры:

```
[
 part1, // например: {"t1 (f11 AS label, f12) AS a1" :
        //   {'ff11 BETWEEN ? AND ?': [v11, v12], ff12: v22}}
 part2, // например: {"t2 AS a2 ON a1.id=a2.id":
        //   {ff21: null}}
 ...
 partN  // например, просто: 'tN'
],
```

Приведённому эскизу запроса соответствует следующий набросок SQL:

```
SELECT
 a1.f11 AS label, a1.f12,
 a2.f21 AS "a2.f21", ...
 ...
 tN.fN1 AS "tN.fN1", ...
FROM
 t_part1 AS a1
 [LEFT|INNER] JOIN t2 AS a2 ON a1.id=a2.id
 ...
 [LEFT|INNER] JOIN tN ON ...
WHERE 1=1
 AND a1.ff11 BETWEEN ? AND ?
 AND ...
[ORDER BY ...]
```

В следующих разделах синтаксис детализирован, но пока уточним область применимости `this.db.query`.

Ограничения

Как видно из представленной схемы, в результате в принципе не предусмотрены:

- вычисляемые выражения и функции;
  - следовательно, агрегаты и `GROUP BY`;
    - тем более аналитика вроде `ROLLUP`
- `UNION`
  - и прочие множественные операции
- вложенные `SELECT` до `FROM`
- табличные выражения во `FROM`
- `WITH` и пр.

Кому не хватает чего-либо из перечисленного (или забытого), но интересен минимализм или остро требуется гибкая генерация запросов на лету — тот волен воспользоваться всей мощью полноценного синтаксиса SQL посредством описания необходимых VIEWs и, возможно, триггеров в модели.

Возможности

В сущности, всё, что умеет этот генератор SQL-кода — это

- распределять каждый объект, связанный с определённой таблицей — часть запроса (part_i) — между разделами `SELECT` / `FROM` / `WHERE`;
- автоматически вычислять `JOIN`'ы по модели там, где это возможно;
- форматировать выражение `WHERE`, параллельно составляя список параметров;
  - игнорируя фильтры со значениями `undefined`, что нужно для форм расширенного поиска в UI.

Анатомия части

Итак, любой запрос для `this.db.query` — это список *частей* минимум из одного элемента. Вначале мы покажем, что представляет из себя *часть* на примерах запросов, где такой элемент всего один (то есть без `JOIN`).

Часть как объект

В общем виде часть представляет собой объект с ровно одним ключом, где:

- ключ — имя таблицы, возможно, со списком полей в скобках;
- значение — объект, представляющий собой набор фильтров.

Пример:

```
[{'vw_ssh_command_items(host, src AS label)': {
 id_command: id,
 'status <>': 'ok'
}}]
```

порождает

```
SELECT host, src AS label
FROM vw_ssh_command_items
WHERE id_command = ? AND status <> 'ok'
```

Список полей можно не указывать — тогда выбираются все столбцы:

```
await this.db.add_all_cnt ({}, [{equipment_cfgs: filter}])

SELECT * FROM equipment_cfgs WHERE — то, что в filter
```

Часть одной строкой

Если с данной таблицей не должно быть связано ни одно условие в `WHERE`, набор фильтров пуст. Тогда вместо объекта можно указать строку, равную значению его единственного ключа:

```
 this.db.add ({}, 'roles')

 SELECT * FROM roles
```

Запросы с такой единственной частью на практике не имеют смысла, однако их применение в `JOIN`'ах вполне может быть обоснованным.

Псевдоним таблицы

Как и в SQL, для таблиц запроса можно указать псевдонимы, воспользовавшись ключевым словом `AS` (в отличие от SQL, здесь `AS` для псевдонимов обязательно):

```
this.db.add_all_cnt ({}, [{users: filter}, 'roles AS role'])
   // ... FROM users LEFT JOIN roles AS role ON ...
```

Строго говоря, псевдоним указывается всегда — просто по умолчанию он совпадает с именем таблицы.

Разумеется, псевдонимы всех частей должны различаться, то есть при использовании одной таблицы более одного раза явное указание псевдонима необходимо.

Подробнее о списке полей

Поля, которые необходимо извлечь в данной части, перечисляются в скобках после имени таблицы. Для каждого из них можно указать псевдоним, по аналогии с таблицей:

```
{'vw_ssh_command_items(host, src AS label)': ...}
   // SELECT host, src AS label ...
```

Если скобок нет, автоматически подставляется `(*)` и выбираются все поля, правда в сгенерированном SQL каждое из них будет упомянуто явно — список берётся из модели.

```
'roles' // SELECT id, label FROM roles
```

Чтобы полностью подавить показ полей из данной части (что имеет смысл в основном для связных таблиц типа "многие-ко-многим"), необходимо явно указать пустой список:

```
    user.opt = await this.db.fold ([
        {'user_options()': {
            is_on: 1,
            id_user: user.uuid
        }},
        'voc_user_options(name)'
    ], (i, d) => {d [i ['voc_user_options.name']] = 1}, {})
```

Имена полей в выборке

Поля первой по порядку (корневой) части фигурируют в результате со своими собственными именами (псевдонимами).

К именам всех остальных полей спереди через точку приписываются имена (псевдонимы) соответствующих частей.

```
        return this.db.add_all_cnt ({}, [
            {vw_tasks : filter},
            'task_notes ON id_last_task_note',
        ])
        SELECT
          vw_tasks.uuid AS "uuid",
          ...
          task_notes.uuid AS "task_notes.uuid",
          ...
```

Фильтры

Объект с набором фильтров, (единственное значение в объекте каждой части) имеет вид:

```
{
 expr_1: values_1,
 expr_2: values_2,
 ...
}
```

Выражения (expr_i) уточняются, дооформляются и копируются в SQL, значения — собираются в список параметров.

Общий вид

В полной форме каждый ключ объекта фильтра представляет собой SQL-выражение, начинающееся с имени поля и содержащее несколько знаков `?`, а значение — список параметров соответствующей длины:

```
{
  'dt BETWEEN ? AND ?': [dt_from, dt_to],
  ...
}
```

Сравнения с константами

В подавляющем большинстве случаев выражение содержит единственную переменную, причём упомянутую в конце. Если это так, знак `?` можно не ставить, а вместо массива значений указать его единственный элемент:

```
{
  'salary >=': threshold,
  ...
}
```

А для выражений, оканчивающихся на `IN`, константы следует задавать (непустым) списком -- тогда будет сгенерировано необходимое выражение с переменными:

```
{
   'id_status IN': [10, 20],
     // {sql: '...id_status IN (?,?)', params: [10, 20]}
}
{
   'id_type NOT IN': ['A', 'B', 'C'],
     // {sql: '...id_type NOT IN (?,?,?)', params: ['A', 'B', 'C']}
}
```

Самый популярный оператор сравнения — равенство (`=`). Его также можно не указывать, что сводит фильтр по фиксированному значению к паре "имя поля-константа":

```
{
  id_status: 10,
  ...
}
```

Взятие по ключу

И, в свою очередь, примерно половина всех фильтров на равенство приходится на первичные ключи соответствующих таблиц. Поскольку ключевое поле известно из модели, здесь можно сэкономить, указав в качестве фильтра не объект, а скалярное значение:

```
let data = await this.db.get ([{users: this.rq.id}, 'roles AS role'])
```

Значения null и undefined

`this.db.query` — редкая функция, где нашлось применение такой экзотической особенности javaScript, как наличие двух разных пустых значений. А именно:

- `null` порождает предикат `IS [NOT] NULL`;
- `undefined` отключает фильтр:

```
{
  'dt_from <>' : null,      // dt_from IS NOT NULL
   dt_to       : null,      // dt_to IS NULL
   id_status   : undefined  // фильтра по id_status не будет
   ...
}
```

`undefined` можно использовать при реализации расширенного поиска в Web-интерфейсах, когда отсутствие значения в поле является требованием игнорировать фильтр по нему. Однако следует соблюдать осторожность: в JSON есть `null`, но нет `undefined`, поэтому нельзя передавать напрямую в фильтр фрагмент тела POST.

Это обстоятельство не создаёт большой проблемы: так или иначе, для обработки параметров, формируемых на клиенте, обычно требуется какой-нибудь переходник. В частности, Dia.pm включает такой модуль для формата, используемого w2ui.

Конструкции IS NULL OR

Нередко в запросе требуется выбрать записи, у которых заданное поле:

- либо пусто;
- либо удовлетворяет заданному (не)равенству.

Типичный пример: поле `dt_to` (дата окончания действия), которая для актуальных документов может находиться в будущем, а может и оставаться пустой. Для удобства описания таких условий в `this.db.query` поддерживается местный синтаксический элемент: многоточие после имени поля:

```
{
  'dt_from  <=': dt,  // AND dt_from <= ?
  'dt_to... >=': dt,  // AND (dt_to IS NULL OR dt_to >= ?)
}
```

Шаблоны для LIKE

При поиске по подстроке текстового поля:

- с клиента приходит сама искомая подстрока,
- а в SQL её необходимо обрамить символами `%` с одной или двух сторон,

что неудобно.

На этот случай в `this.db.query` предусмотрен "синтаксический подсластитель": возможность приписать `%` не к значению, а к переменной, то есть в текстовый шаблон фрагмента запроса. Эти символы будут вычищены из SQL, но добавлены к значениям параметров:

```
{
   'inn   LIKE ?%' :  77,
     // {sql: '...inn   LIKE ?', params: ['77%']}
}
{
   'label LIKE %?%': 'dia',
     // {sql: '...label LIKE ?', params: ['%dia%']}
}
```

В принципе данный механизм не привязан к ключевому слову `LIKE` и будет работать в любых ситуациях, когда со знаком вопроса соседствуют проценты. Это может иметь смысл для расширений различных диалектов SQL: например, `ILIKE` в PostgreSQL.

Подзапросы

`this.db.query` позволяет использовать оператор `IN` не только со списками констант, но и с вложенными SQL-запросами. Они должны быть представлены объектами вида `{sql:..., params:...}`. При таком типе параметра оператор `IN` подставится автоматически

```
{
  id_type: {sql: 'SELECT id FROM types WHERE kind = ?', params: [1]}
  // ...WHERE id_type IN (SELECT id FROM types WHERE kind = ?)
}
```

Поскольку результат самой функции `this.db.query` имеет вид `{sql:..., params:...}`, её можно использовать для формирования вложенных подзапросов:

```
{
  id_type: this.db.query ({'types(id)': {kind: 1}})
}
```

Псевдофильтры

Две компоненты реестра фильтров с фиксированными именами имеют смысл, отличный от ограничения значений полей. Обе они имеют смысл только в одной -- первой -- части запроса

ORDER

Значение, соответствующее ключу `ORDER`, копируется в раздел `ORDER BY` результирующего SQL:

```
this.db.list ([{task_notes: {
  id_task: this.rq.id,
  ORDER: 'ts'
}}]),
// SELECT * FROM task_notes WHERE id_task = ? ORDER BY ts
```

LIMIT

По этому ключу может быть записан либо массив из одного или двух натуральных чисел:

```
{
 LIMIT: [50, 0],
 //...
}
```

либо отдельное числовое значение:

```
{
 LIMIT: 1,
 //...
}
```

Заданные здесь значения не влияют на генерируемый текст SQL, а копируются в компоненты `limit` и `offset` результирующего объекта, откуда они, наряду с `sql` и `params`, используются при вызове this.db.add_all_cnt.

Соединение частей

В этом разделе показано, как в аргументе `this.db.query` оформляются условия для выражений `JOIN` раздела `FROM`.

LEFT или INNER

По умолчанию для всех присоединяемых таблиц `this.db.query` генерирует конструкцию `LEFT JOIN`.

Если необходимо использовать `INNER JOIN`, перед названием соответствующей таблицы следует поставить знак `$`:

```
 return this.db.add_all_cnt ({}, [
  {task_notes: filter},
  {'$tasks(uuid, label) ON task_notes.id_task': task_filter}
 ])
 SELECT ... FROM task_notes
  INNER JOIN tasks
   ON (task_notes.id_task = tasks.uuid AND /* см. task_filter */)
```

Явное указание условия связи

При необходимости (что случается довольно редко) условие, которое должно фигурировать после `JOIN ... ON`, можно привести дословно:

```
return this.db.add ({}, [
    {'users(uuid, label, uuid AS id)': {
        'login <>' : null,
        'uuid  <>' : this.user.uuid,
        is_deleted : 0,
        ORDER      : 'label',
    }},
    {'user_users AS user_user ON user_user.id_user_ref = users.uuid': {
        is_on: 1,
        id_user: this.user.uuid,
    }}
])
SELECT ... FROM users
  LEFT JOIN user_users AS user_user
    ON user_user.id_user_ref = users.uuid
```

Признак того, что условие приведено полностью — наличие знака `=` в ключе объекта, представляющего вторую часть. В этом случае `this.db.query` не пытается вычислить условие, а копирует его как есть.

Указание единственного поля-ссылки

В подавляющем большинстве случаев условие связи сводится к равенству между полем-ссылкой одной таблицы и первичным ключом другой. Поскольку информация о ссылках имеется в модели, для такой связи достаточно указать лишь одно поле (если его имя единственно для ранее упомянутых частей запроса — иначе приходится писать условие в явном виде).

```
 return this.db.add_all_cnt ({}, [
   {vw_tasks : filter},
   'task_notes ON id_last_task_note',
 ])

 SELECT ... FROM vw_tasks
   LEFT JOIN task_notes
     ON vw_tasks.id_last_task_note = task_notes.id
```

Автоматическое вычисление связи

Нередки случаи, когда для заданного множества таблиц `JOIN` вообще однозначно вычисляется без явного указания чего-либо: просто на основе анализа набора полей-ссылок.

```
 this.db.add_all_cnt ({}, [{users: filter}, 'roles AS role'])
 SELECT * FROM users
  LEFT JOIN roles
   ON users.id_role = roles.id
     // потому что других ссылок на roles нет
```

Маршрутизация запросов

Общие слова

Любой HTTP-сервер математически представляет собой вычислитель функции. Аргумент: строка (HTTP-запрос), значение — строка (HTTP-ответ).

```
response = handler (request)
```

Понятие о маршрутизации

На практике удобнее считать, что вычисляется не одна-единственная функция, а одна из множества — в зависимости от некоторых свойств запроса. То есть:

- сначала вычисляется значение функции-маршрутизатора (назовём её `router`);
- затем, в зависимости от полученного значения, с полученным аргументом вычисляется одна из функций-обработчиков (назовём её `handler`):

```
response = handler (router (request)) (request)
```

Структура HTTP-запроса содержит множество компонент, которые используют маршрутизаторы:

- глагол (GET, POST и т. п.)
- адрес (URL), в том числе:
  - последовательность отрезков пути (path);
  - параметры запроса (query string): анонимный и именованные;
- заголовки: стандартные и придуманные по месту;

- наконец, содержимое тела запроса (при допускающих его наличие глаголах: типа POST и PUT).

Частные случаи

Рассмотрим два популярных подхода к маршрутизации HTTP-запросов.

SOAP/HTTP

Спецификация SOAP — хороший пример детально проработанного документа. Что неудивительно, поскольку писалась она в 2000-м году людьми из IBM, Lotus и Microsoft. Тем не менее, и там пункт, непосредственно касающийся HTTP, изложен какими-то водянистыми намёками:

- POST — единственный описанный глагол, но не исключены и другие,
- `SOAPAction` обозначает *намерение* (intention) и может иметь произвольное значение.

Тем не менее, на практике сложилось достаточно однозначное толкование, согласно которому:

- все запросы имеют глагол POST;
- URL определяет *сервис* (в сущности, пакет функций);
- имя конкретной функции в пакете приличные люди указывают заголовком `SOAPAction`;
- а все прочие (кого не интересуют, например, проблемы проксирующих систем) предоставляют угадывать нужную функцию по XML-типу аргумента — что, соответственно, возможно только при разборе тела POST.

Это вполне работоспособный вариант, в особенности для естественной среды применения SOAP: интеграции с заклятыми врагами.

Однако здесь немного не хватает представления о классификации функций-обработчиков. Методы: извлекающие, генерирующие, модифицирующие данные — каждый из них может называться, как угодно. В любом проекте, где таких методов сотни, непременно возникают местные соглашения об именовании (например, префиксы set- и get-, export- и import- и т. п.), но в общей спецификации намёка на это нет, так что каждый сервис может оформляться по-своему.

«REST»

Кавычки в заголовке этого раздела поставлены специально: по аналогии с названием одного известного музыкального произведения. Мы не претендуем на рассказ о правилах истинного R.E.S.T. Это совершенно бесперспективное занятие, поскольку, в отличие от SOAP, спецификации REST нет и никогда не будет.

Однако в общественном сознании объективно существует расхожее понятие о "рестах", которое сводится к тому, что:

- глаголы POST/GET/PUT/DELETE соответствуют действиям create / read / update / delete (жизненный цикл CRUD);
- путь у URL имеет вид `/{type}/` для коллекций или `/{type}/{id}` для отдельных записей.

Такой подход лишён недостатков, упомянутых в прошлом разделе, однако в непосредственном виде применим лишь к простейшим задачам, ограниченным пресловутым CRUD.

Однако лишь речь заходит о реализации процессов с десятком действий вроде "утвердить", "аннулировать", "подправить", "перезапустить" и т. п. — глаголы очень быстро заканчиваются и выясняется, что RFC 7231 не резиновый.

И начинается подбор костылей с оформлением лишних действий:

- либо в виде компонент пути (что рушит модель "один ресурс — один путь");
- либо в виде параметров строки запроса (что гораздо прозрачнее на уровне HTTP-трафика, но не так удобно для типично "рестовских" библиотек маршрутизации).

И тут же пропадает всё обаяние "рестов", столь привычное по примерчикам уровня "Hello world", изложенным на сайтиках категории single page, оптимизированных под флагманские планшеты.

Конкретика

В Dia.js применяется маршрутизация запросов, весьма близкая к вышеупомянутому «REST», однако лишённая привязки к невосполнимому ресурсу списка HTTP-глаголов и связанных с этим проблем масштабируемости. Кроме того, в отличие от «REST», не ставится задача сделать URL максимально приятным человеческому глазу, поскольку наперёд известно, что обрабатывать его будет машина.

Итак, изложим простые правила. В Dia.js принят следующий формат URL запросов (безотносительно к глаголу):

```
{root}?[type=...][&id=...][&action=...][&part=...]
```

, где `{root}` — общий корневой путь (по аналогии с SOAP/HTTP), у Dia-приложений обычно имеющий значение `/_back/`.

Имена следующих четырёх параметров фиксированы:

type

> Тип данных: как 1-я компонента пути в «REST». Присутствует почти во всех запросах. Запрос без `type` обычно лишён и всех остальных параметров и предназначен для продления сессии Web-интерфейса. Для запросов, явно связанных с редактированием записей в определённой таблице БД, как правило совпадает с именем этой таблицы.

id

> Идентификатор объекта: как 2-я компонента пути в «REST». Пуст у запросов извлечение коллекций данных либо на массовые преобразования коллекций, не связанные с каким-либо одним объектом данных. Обычно совпадает по значению с первичным ключом некоторой записи в одной из таблиц БД (если `type` — имя таблицы, то в ней).

action

> Наименование действия — по смыслу соответствует HTTP-глаголу в «REST». Пусто у запросов на выборку данных. Для стандартных действий имеет значения из фиксированного списка (`create`, `update`, `delete`). Для прочих действий значения `action` подбираются согласно правилам именования, принятым в конкретном проекте. Как правило запрос с непустым `action`, порождает транзакцию в БД.

part

> Имя раздела данных. Присутствует в запросах на извлечение информации, где требуется не вся полнота данных, связанных с коллекцией или объектом, а только её специфический срез. Данный параметр имеет вспомогательное значение и редко применяется, поэтому в целом в настоящей

документации говорится о системе координат type/id/action без его упоминания.

Сторонние форматы

Разумеется, фиксировать набор параметров можно только в рамках приложений собственной разработки (свой сервер — свой же клиент), а при интеграции необходимо принимать всё, что пришлют коллеги.

Модель type-id-action не ставит в этом плане принципиальных ограничений. Для любого конкретного стороннего «REST»'а, SOAP или иного интерфейса несложно создать обработчик, который при первичном разборе будет выполнять дополнительное преобразование и вычислять значения трёх основных параметров из произвольных компонент полученного запроса.

В качестве примеров можно указать:

- EluStatic, выдающий файлы с диска в ответ на GET-запросы;
- класс JsonRpc, который разбирает тело полученного POST'а согласно спецификации JSON-RPC 2.0 и вычисляет извлекает оттуда значение атрибута `method` как `action`, а `type` берёт из корня URL — по аналогии с «REST».

Предварительная маршрутизация

Когда в одном приложении HTTP-запрос, поступивший на некоторый порт, может быть передан различным обработчикам, встаёт вопрос о том, как оформить в коде соответствующее решающее правило.

Предлагаемое решение: взять класс HTTP/Router и доопределить его метод `create_http_handler` с учётом специфики проекта.

Пример того, как это выглядит для разделения запросов на статику и динамику, приведён в коде эталонного приложения.

Не только HTTP

Dia.js обрабатывает запросы разных форматов и разной природы. Они не обязательно должны поступать в виде HTTP-запросов, но могут приходить по e-mail, системам мгновенных сообщений, а также AMQP, 0MQ и прочим подобным каналам.

Однако в любом случае для каждого поддерживаемого формата сообщений определён способ указания по крайней мере трёх важнейших параметров `type`, `id` и `action`.

В зависимости от их значений обработчик принимает решение о том, какой загрузить модуль и какой функции в нём передать управление.

Обработчики

Во многих приложениях из реальной жизни львиная доля реализованных там бизнес-операций содержат ряд типовых шагов, не зависящих от входных данных: вычисление текущего пользователя по номеру сессии, подключение к БД и т. п.

В Dia.pm такого рода логика реализуется в определённом слое: классах *обработчиков* запросов.

Обработчик в Dia.js -- это одноразовый объект для обслуживания каждого отдельного запроса.

Экземпляры обработчиков создаются:

- для внешних запросов (в частности, HTTP) -- прослушивателями событий, инициируемыми в index.js;
- для подзапросов -- фунциями из модулей, реализующих бизнес-логику приложения.

Жизненный цикл любого обработчика сводится к:

- инициализации;
- исполнению асинхронного метода `run`, по ходу которого:
  - расшифровываются входные параметры (например, из тела POST-запроса);
  - резервируются необходимые ресурсы (связь с БД, почтовым сервером и т. п.);
  - определяется необходимый модуль и метод в нём;
  - выбранный метод вызывается в контексте текущего обработчика, доступного там как `this`;
  - ресурсы освобождаются;
  - полученный результат либо ошибка оформляются в соответствии с природой обработчика (генерируется HTTP-ответ с тем или иным кодом, вызывается `resolve` либо `reject` для Promise и т. п.)

Обработчики ядра

Классы всех обработчиков в приложениях на базе Dia.js восходят к общему предку Handler, в том числе:

- обработчики HTTP-запросов -- к одному его наследнику HTTP;
- обработчики подзапросов -- к другому: Async.

Обработчики приложения

Классы обработчиков, входящие в Dia.pm, в частности HTTP, предназначены не для прямого использования в коде приложения, а для уточнения. В структуре директорий для них отведён каталог back/lib/Content/Handler.

Простейшее приложение, представляющее собой Web-интерфейс редактирования учётных записей пользователей, содержит единственный класс обработчика, унаследованный от HTTP.

Двойное наследование

В чуть более сложном приложении -- системе учёта заданий -- имеется уже используется два класса обработчиков разной природы: WebUiBackend для входящих HTTP-сообщений и Async для подзапросов.

Некоторые методы, реализующие верхнеуровневую логику приложения, для них необходимо переопределить единым образом: в частности, `get_method_name`, реализующий соглашения об именовании функций в модулях. Естественно было создать класс Base и вынести общий код в него.

Но вот вопрос: как организовать наследование от Base, если рассматриваемые обработчики являются наследниками разных классов, которым Base в предки решительно не годится? К счастью, в javaScript ООП не догма, так что можно впрыснуть любую функцию в качестве метода готовому классу так же легко, как подсунуть что угодно в качестве переменной `this`.

В базовом Handler определён метод `import`, предназначенный для копирования методов из посторонних классов. Вот как это выглядит в упомянутом приложении:

```
constructor (o) {
    super (o)
    this.import ((require ('./Base')), ['get_method_name', 'fork'])
}
```

Выдача статических файлов

EluStatic -- это обработчик HTTP-запросов, представляющий собой сервер статических файлов крайне ограниченной функциональности: он предназначен для выдачи HTML/CSS/js Web-приложений, написанных на базе elu.js.

В норме на любой рабочей (и приближенной к ней тестовой) среде серьёзного Web-приложения запросы на статические файлы должны бы обслуживаться специально оптимизированным для этого сервером (таким, как nginx). Инструкция по установке типового приложения описывает такую конфигурацию.

Однако на этапе разработки производительность не так важна, как простота установки и конфигурации. Кроме того, в некоторых приложениях с немногочисленными пользователями UI (один администратор, заходящий раз в месяц) простой встроенный сервер статики также может пригодиться.

`EluStatic` написан для приложений со стандартной структурой директорий, ему не требуется никакая настройка и он не принимает никаких параметров. Корневой каталог статических файлов определяется относительно директории текущего исполняемого файла, то есть index.js. Для приложений с более специфической структурой можно либо определить класс, наследующий `EluStatic`, либо написать свой аналогичный обработчик -- смотря, что окажется проще и быстрее.

`EluStatic` выдаёт ответы только с двумя кодами:

- 200 OK, если нашёл файл по URL (с учётом rewrite rule по поводу номера сборки статики);
- 404 File Not Found в противном случае.

HTTP-заголовок `Content-Type` вычисляется только для нескольких расширений файлов, используемых в elu-приложениях. Для HTML и js объявляется кодировка `utf-8` (что оказалось невозможно средствами стандартного node-static -- одна из причин, по которым EluStatic написан с нуля).

Для файлов, пути которых содержат номер сборки, добавляется `Expires` в далёком будущем. Соответственно, они кэшируются на клиенте до смены версии.

Одно и то же Dia-приложение будет одновременно работать как в комплекте с nginx (в качестве backend'а для реверсного proxy), так и без него -- принимая обращения на свой порт прямо из браузера.

С точки зрения Dia.js EluStatic представляет собой вырожденный случай обработчика в том смысле, что он не использует ни конфигурацию, ни ресурсы, ни модули. С другой стороны, он же демонстрирует гибкость модели, реализуя простое решение реальной задачи, не входя в конфликт с ненужными для этого особенностями платформы.

Подзапросы

Обработчик файлов

Модули

Модуль в Dia.pm — это набор функций для обработки запросов, связанных с определённым **типом** (то есть одним из значений `type` в модели type-id-action).

В структуре директорий для них отведён каталог back/lib/Content.

Внутри этой директории располагаются модули node.js, каждый из которых экспортирует "объект js", то есть набор пар вида:

- ключ — имя метода
- значение — функция, реализующая метод.

Какой модуль будет и какой его метод использован для обработки запроса — решает обработчик.

По умолчанию имя (файла) модуля совпадает со значением параметра `type`.

Выбор имени метода реализуется в классе обработчика конкретного проекта. В типовом приложении реализована логика, в основном унаследованная со времён Eludia.pm:

- если указан только `type` — вызывается `select_${type}` (например: `select_users`);
- если указан только `type` и `id` — `get_item_of_${type}` (например: `get_item_of_users`);
- если указано `action` — `do_${action}_${type}` (например: `do_update_users`);
- и если указан `part` — `get_${part}_of_${type}` (например: `get_vocs_of_users`).

Указание `type` в названии каждого метода, принадлежащего к одноимённому модулю, может показаться избыточным. Но на практике это не мешает, а, наоборот, позволяет быстрее сориентироваться в коде.

По ходу исполнения каждого метода в качестве переменной `this` фигурирует вызвавший его обработчик. То есть в момент вызова функция, определённая в модуле, осознаёт себя методом объекта-обработчика.

При этом родной модуль виден ей как `this.module` — таким образом можно получить доступ к соседним методам. Правда, если делать это напрямую, надо не забыть обеспечить контекст исполнения (тот же `this`). Удобнее воспользоваться специальным методом `this.call (sibling_method_name)`. В примере ниже при выдаче учётной записи `get_item_of_users` пользователя подгружается фиксированный набор справочников `get_vocs_of_users`, который может быть получен и отдельным AJAX-запросом (для отображения реестра):

```
get_item_of_users:
    async function () {
    let [data, vocs] = await Promise.all ([
        this.db.get ({users: this.rq.id}),
        this.call ('get_vocs_of_users'),
    ])
    return Object.assign (data, vocs)
    },
```

Один и тот же метод может подставляться на лету в обработчики разных классов. Что позволяет использовать его для работы с сообщениями,

пришедшими по разным каналам: например, генерировать по общим правилам ответы как на HTTP-запросы, так и на AMQP-сообщения. В мире J2EE это бы выглядело как объект, одновременно являющийся сервлетом и MDB (если бы такое было возможно).

Переменная `this` предоставляет методу API, необходимый для реализации бизнес-логики.

Значение, вычисленное методом модуля поступает на вход методу `send_out_data` текущего обработчика и как конкретно будут использованы эти данные — зависит от природы последнего. В частности:

- HTTP:
  - объекты типа stream.Readable — передаёт напрямую в поток ответа;
  - прочие объекты — оборачивает в JSON и, сопроводив соответствующими заголовками, тоже выдаёт в ответ.
- Async:
  - передаёт полученное значение `resolve`-функции.

API

Запуская метод модуля, обработчик передаёт ему ссылку на себя в качестве переменной `this`. Таким образом для реализации бизнес-логики метода становятся доступны:

- this.uuid — уникальный номер запроса;
- this.conf — объект конфигурации приложения (в Eludia.pm это была переменная `$preconf`);
- this.rq — объект с параметрами запроса (в Eludia.pm это называлось `%_REQUEST`, как `$_REQUEST` в PHP);
- this.user — объект с реквизитами текущего пользователя (аналог `$_USER` из Eludia.pm);
- для каждого ресурса — одноимённая компонента
  - в частности, если заявлен БД-ресурс с именем `db` — this.db (как `$db` в Eludia.pm — но здесь это имя не фиксировано равноправных БД может быть много).

this.uuid

Каждый запрос в Dia.js имеет уникальный номер: случайное число в формате UUID.

Не стОит путать его с параметром `id` из тройки type-id-action, который также вполне может быть UUID'ом:

- `id` (доступный как `this.rq.id`) -- это (как правило) уникальный ключ хранимой бизнес-сущности;
- `this.uuid` -- это одноразовый номер самого запроса, предназначенный, прежде всего, для отслеживания истории процесса по log-файлу.

Типичное применение `this.uuid` -- выбросить его на клиент в случае фатальной ошибки, чтобы UI отобразил готовое обращение в техподдержку. Так гораздо быстрее искать корни проблем, чем при белом экране. Описанный механизм реализован на уровне стандартного обработчика HTTP. Если клиент использует elu.js, то нужные сообщения оформляются автоматически.

this.rq

Автор этих строк не относится к числу поклонников PHP, однако одна находка там, безусловно, заслуживает уважения. Это наличие переменной `$_REQUEST`. Глобалка с естественным именем и очевидным содержимым, которое гарантировано безотносительно к деталям HTTP-трафика.

Ровно такая же переменная в API Dia.js -- `this.rq`. Только с небольшой поправкой на то, что в нашем 2019 году часто называется "REST"ом.

Базовый HTTP-обработчик, получив запрос с телом типа "application/json", распаковывает его в `this.rq`, а потом укладывает туда на верхнем уровне ключи/значения, расшифрованные из строки поискового запроса (query string): в норме там должны быть только type-id-action, но могут быть и другие параметры.

Таким образом, запрос

```
POST /_back/?type=users&id=1&action=update
...
{"data": {"label":"admin"}}
```

будет распакован в

```
this.rq = {
  type: 'users',
  id: '1',
  action: 'update',
  data: {
    label: 'admin'
  }
}
```

Верхнеуровневый объект `data` (или `search` или ещё какой-нибудь) рекомендуется включать в тело POST во избежание путаницы.

Классы-наследники могут реализовывать свои правила, переопределяя метод `read_params`, но суть должна оставаться той же: `this.rq` -- объект, содержащий полный набор параметров запроса.

this.user

Общие слова

Поле обработчика `user`, доступное на этапе исполнения метода модуля как `this.user`, предназначено для представления данных о пользователе, от имени которого совершается текущая операция.

Если приложение хранит учётные записи своих пользователей, в таблице БД (типовое имя — `users`), то обычно `this.user` по набору полей и их содержимому соответствует одной из записей этой таблицы, по крайней мере частично. Такую логику средствами Dia.js легко реализовать, однако на уровне ядра системы она не зафиксирована.

Конкретика

До вызова метода обработчик делает, упрощённо говоря, следующее:

```
if (!this.is_anonymous ()) {
  this.session = this.getSession ()
```

```
  this.user    = this.getUser    () // this.session.getUser ()
}
```

По умолчанию `is_anonymous ()` возвращает `true` и оба значения получаются пустыми. Если это требуется исправить, нужно добавить немного логики на уровне приложения, реализовав местное представление о сессии.

Сессия в Dia.pm — это объект:

- возвращаемый методом `getSession ()` обработчика;
- реализующий методы:
  - `start ()`;
  - `getUser ()`;
  - `finish ()`.

В качестве переменной API сессия (в отличие от ресурсов) предполагается не долгоживущей, а столь же одноразовой, как сам обработчик. Собственно говоря, это часть обработчика, вынесенная в отдельный объект для облегчения читаемости кода.

В структуре директорий файлы классов сессий, связанных с протоколом HTTP, должны располагаться в back/lib/Content/Handler/HTTP. Для многих используемых на практике схем аутентификации можно воспользоваться имеющимися в ядре Dia.pm базовыми классами.

Аутентификация и выход

В норме приложение, предполагающее собственную аутентификацию пользователей, должно содержать модуль `sessions` с методами следующего вида:

```
do_create_sessions: async function () {
  // ... проверки и прочие вычисления ...
  this.user = // ... откуда его брать
  await this.session.start ()
},

do_delete_sessions: async function () {
  await this.session.finish ()
}
```

Значение, присвоенное `this.user` перед `this.session.start ()` далее будет доступно в качестве `this.user` на входе в каждый метод в серии связанных (например, cookie) запросов, пока либо не будет вызвано `this.session.finish ()`, либо со времени запроса не истечёт время, устанавливаемое в реализации `getSession ()` текущего обработчика — обычно это тот или иной параметр конфигурации.

Подробнее об анонимности

Описанный разделом выше `do_create_sessions` -- такой же метод приложения, как и все прочие. Соответственно, по умолчанию до его вызова обработчик попытается установить текущего пользователя. Чтобы предотвратить этот абсурд, нужно соответствующим образом переопределить у обработчика упомянутый ещё раньше метод `is_anonymous ()`:

```
is_anonymous () {
  return this.rq.type == 'sessions' && this.rq.action == 'create'
}
```

Так выглядит данный метод в обработчиках Web-приложений, требующих обязательной аутентификации для всех запросов (кроме самого login'а). Для сайтов с публичной частью условие может быть гораздо слабее, а обработчики запросов mashine-to-mashine могут вообще игнорировать понятие "пользователя" и, тем более, механизм сессий.

Если `is_anonymous ()` всё-таки вернул `false`, то, как мы знаем, вызывается `getSession ().getUser ()`. Если объект сессии не может вычислить пользователя по параметрам запроса, в норме он должен вызвать метод обработчика `no_user ()` и вернуть его результат. У обработчика HTTP это приводит к выбросу 401 Unauthorized, но у его наследников может быть переопределено, например, на выдачу специальной анонимной учётной записи.

Схемы аутентификации

Ядро Dia.js содержит следующую династию классов, реализующих стандартные схемы аутентификации:

- Session — сферическая сессия в вакууме: нечто, имеющее номер и связанное с обработчиком;
  - CookieSession — то, что умеет добывать у обработчика HTTP-заголовок `Cookie` и разбирать его, а также формировать `Set-Cookie`;
    - CachedCookieSession — то, что умеет использовать полученное значение cookie в качестве ключа в кэше.
    - JWTCookieSession — то, что трактует значение cookie как JSON Web Token.
  - BasicSession — реализация схемы Basic;
  - BearerSession — базовая реализация схемы OAuth 2.0 (`Authorization: Bearer`);
    - JWTBearerSession — конкретизация `Authorization: Bearer` для значений в формате JSON Web Token.

Перечисленные классы и их потомки предназначены для формирования значений метода `get_session` обработчика.

Основное, что должен делать каждый такой объект-сессия: выдавать методом `get_user` текущую учётную запись пользователя -- обработчик сделает её видимой методу модуля в качестве переменной `this.user`.

Внутренняя аутентификация

В данном разделе описаны схемы, предполагающие создание и завершение сессий в рамках приложения (в отличие от Single Sign On и т. п.). На момент написания данного текста все они используют механизм cookies, поэтому соответствующие классы восходят к `CookieSession`.

Они осмысленно реализуют методы `start`, `keep_alive` и `finish`, действие которых выражается в установке `Set-Cookie` и обновлении записи в хранилище на сервере (если таковое применяется).

Хранимые сессии с синтетическими id: CachedCookieSession

В Dia.js реализованы два класса для хранения записей в течение фиксированного времени:

- MapTimer — кэш в памяти, использующий стандартные таймеры javaScript;
- Memcached — переходник к драйверу одноимённого ПО.

Оба они, хотя могут использоваться в разных контекстах, непосредственно предназначены для стыковки с `CachedCookieSession`:

- `MapTimer` доступен всегда, но непригоден для масштабируемых решений;
- `Memcached` реализует централизованное хранение сессий, заведённых в разных адресных пространствах, но, разумеется, требует доступа с установленному Memcached.

Использовать тот, другой или ещё какой-то класс для хранения сессий — в норме должно быть предметом *настройки* приложения на конкретной среде. Делается это так:

- при загрузке конфигурации создаётся и объявляется ресурсом кэш нужного типа;
- местный класс обработчика в своём методе `getSession` использует этот ресурс, не вдаваясь в детали реализации.

Как это организовать — показано в типовом приложении.

По аналогии с Memcached несложно реализовать аналогичные решения для иных хранилищ данных -- однако это представляется целесообразным только при наличии там функции ограничения времени хранения записей (как, например, в Redis). Пока автору не известно о подобной функциональности в реляционных (SQL) СУБД, поэтому соответствующий вариант хранения сессий в Dia.js отсутствует.

Идентификатор в виде УЗ с подписью: JWTCookieSession

Согласно RFC 7519, JSON Web Token предназначен для передачи в URL или в HTTP-заголовке `Authorization`.

Тем не менее, имеются веские причины использовать в Web-интерфейсах (в отличие от (микро)-сервисов mashine-to-mashine) другой заголовок: `Cookie`.

Данная схема реализована в виде класса JWTCookieSession.

Он обеспечивает передачу значения `this.session.user` в виде поля `sub` тела токена. Поскольку в данном случае учётная запись пользователя (+ электронная подпись) передаётся практически открытым текстом, то, в отличие от CachedCookieSession, вопрос о дополнительном хранилище данных не возникает и соответствующего поля нет. Зато добавляются методы получения ключей шифрования:

```
async get_private_key () {
  //...
}

async get_public_key () {
  //...
}
```

Внешняя аутентификация

В этом разделе описаны классы сессий, порождаемых вне приложения: для них методы `start` и `finish` не имеют смысла.

Такие "сессии" с ограниченной функциональностью соответствуют либо сценариям Single Sign On в web-интерфейсах, либо прямому указанию учётной записи при вызове Web-сервиса.

BasicSession

В 2019 году мало в каком web-интерфейсе можно увидеть формочку старого доброго Basic'a. Однако при организации доступа к сервисам mashine-to-mashine (где по идее должны бы быть сертификаты) он встречается сплошь и рядом. Для поддержки данной схемы на уровне приложения в Dia.pm предусмотрен класс BasicSession.

Вот что делает BasicSession:

- проверяет наличие заголовка `Authorization: Basic...`;
- при его отсутствии -- выдаёт 401 Unauthorized;
  - при этом используется опция `realm`, по умолчанию ей приписывается значение `REQUIRED`;
- а при наличии:
  -
    - иначе устанавливает значения своих полей `user` и `password`.

ВНИМАНИЕ! Если использовать этот класс без переопределения `get_user`, то обработчик установит `this.user` равным строке login'a вообще без какой-либо проверки пароля. Автор исходит из того, что если уж люди передают пароль открытым текстом (Base64 -- не в счёт), то вряд ли их убережёт какой-то автомат на уровне ядра.

С другой стороны, организовать проверку пароля, например, с использованием типового механизма шифрования стоит всего 3 строки кода.

JWTBearerSession

Класс JWTBearerSession реализует очень похожую схему аутентификации, только вместо слова `Basic` использует `Bearer` (см. RFC 7617, 6750) и рассматривает полученную строку не как пару `user:password`, а как значение упоминавшегося выше JSON Web Token.

Как и в случае `JWTCookieSession`, здесь на выходе стандартного `get_user` фигурирует декодированное значения поля `sub`.

Пароли

Общие слова

Начнём с нескольких банальностей. Прежде всего: при малейшей возможности следует избегать хранения любых конфиденциальных данных и выполнения каких-либо криптографических расчётов в своей системе.

Если оказывается необходимо проверять предъявленные на вход пароли, которые никто не требует передавать вовне -- хранить надо не сами пароли, а их контрольные суммы (дайджесты).

Есть плохие, негодные алгоритмы расчёта дайджестов (как OLD_PASSWORD в MySQL), но достаточно надёжных не может существовать в принципе. В том смысле, что сколько ни перемешивай нули с единицами детерминированным образом -- всё равно результат этой операции для строки 'qwe123' и всех подобных можно получить прямо в поисковике, а прочих -- за разумные деньги в радужных таблицах.

Единственное, что может спасти от обращения функции через перечисление известных результатов -- это добавление к аргументу дополнительного содержимого (то есть "соли"). Достаточной длины, чтобы вывести результат из множества просчитанных.

Однако если приписывать ко всем паролям пусть очень длинный, но фиксированный фрагмент, то для одинаковых паролей будут получаться одинаковые значения -- а это позволяет, например, находить учётки с совпадающими паролями. Поэтому есть смысл добавлять к паролю два значения "соли": детерминированное и случайное (запоминая второе тут же рядом).

Dia.js содержит класс-калькулятор контрольных сумм, который помогает использовать в приложении описанную схему.

Он добавляется в конфигурацию приложения в качестве ресурса

```
this.pools = {
 //...
 pwd_calc: new (require ('./Ext/Dia/Crypto/FileSaltHashCalculator.js')) ({
   salt_file: this.auth.salt_file,
   // algorithm: 'sha256', // <-- default value
   // encoding:  'hex',    // <-- also default; set null to get a Buffer
 }),
}
```

и, соответственно, становится виден в коде приложения как `this.pwd_calc`. Теперь в коде сессии (или самого обработчика -- как в каком приложении удобнее) можно добавить метод:

```
async password_hash (salt, password) {
 return this.h.pwd_calc.encrypt (password, salt)
}
```

Он шифрует пароль с учётом специфики данного проекта: берёт ключевой файл в известном месте, использует местные опции. Остаётся только применить этот метод:

- при записи пароля;
- и при его проверке.

Ресурсы

Ресурсами называются объекты, используемые при обслуживании более, чем одного запроса -- в том числе нескольких параллельных запросов. В основном это долгоживущие сетевые соединения со сторонним ПО. Примеры ресурсов:

- подключение к БД;
- подключение к SMTP-серверу;
- подключение к memcached.

При работе с ресурсами важны понятия источника и экземпляра:

экземпляр ресурса

: это переменная, которую можно непосредственно использовать при реализации бизнес-логики (например, соединение с БД);

источник ресурса

: это переменная, предоставляющая синхронный доступ к экземпляру (соответственно, pool соединений).

При загрузке конфигурации все источники ресурсов приложения должны упоминаться в разделе `pools` -- таким образом, в прикладном API они видны как `this.conf.pools`. Однако методы модулей должны обращаться к ним только в исключительных случаях, поскольку им нужны не источники, а экземпляры.

Получением экземпляров (в том числе ожиданием в случае нехватки pool'а) и их последующей безопасной утилизацией в норме занимается обработчик. Для этого у него есть методы `acquire_resources` и `release_resources`, которые, во избежание утечек, не стОит переписывать в приложении.

Отметим, что обработчик резервирует не все подряд ресурсы, объявленные в общей конфигурации, а только те, которые упомянуты в параметре `pools` его конструктора. Для простого приложения может быть достаточно единственного универсального обработчика с полным доступом ко всему. Однако сложная логика и жёсткие требования производительности делают осмысленным выделение нескольких отдельных обработчиков с ограниченными наборами ресурсов. Пример: предназначенный для тысяч параллельных запросов обработчик, лишённый доступа к БД -- и, соответственно, не стеснённый величиной её pool'а (типичная величина: 10).

У каждого необходимого источника обработчик ищет метод `acquire`. Если таковой находится, обработчик вызывает его и запоминает полученный экземпляр в списке на обязательное (finally) освобождение. Если же метода `acquire` нет, значит, нет и проблем с конкурентностью, то есть pool'инг не нужен и экземпляр совпадает с источником.

Экземпляр каждого полученного ресурса добавляется в объект обработчика под тем же именем, с которым источник фигурирует в `pools`. Важнейший пример: экземпляр соединения, объявленного как `conf.pools.db`, будет доступен как `this.db`. Вот с этой переменной и должен работать прикладной код.

this.db — родная БД

Прежде всего, оговоримся: в отличие от прочих компонент API Dia.js, `this.db` — не фиксированное имя переменной (как, например, this.user), а лишь типовое обозначение ресурса определённой природы: связи с реляционной СУБД. Для начала приведём список основных методов API, связанных с БД:

- Высокий уровень (объекты)
  - Выборка данных
    - this.db.query — генератор SQL
    - this.db.fold — итерация / сбор данных
    - Извлечение самостоятельных объектов
      - this.db.get — получение отдельной записи
      - this.db.list — получение полного списка
    - Приписывание с существующим объектам
      - this.db.add — добавление полного списка как компоненты в заданный объект
      - this.db.add_all_cnt — добавление усечённого списка и счётчика записей как компонент в заданный объект
      - this.db.add_vocabularies — добавление словарей-справочников
  - Правка данных
    - Явная
      - this.db.insert — добавление записи
      - this.db.load — массовое добавление записей (пока только postgresql)
      - this.db.update — обновление записи
      - this.db.delete — удаление записей
    - Условная

        - this.db.upsert — добавление/обновление записи
        - this.db.delepsert — добавление/обновление/удаление пакета записей
        - this.db.insert_if_absent — добавление записи при условии отсутствия
- Низкий уровень (SQL как строка)
    - Извлечение данных
        - this.db.select_all — получение списка
        - this.db.select_all_cnt — получение части списка и общего числа записей
        - this.db.select_scalar — получение отдельного поля
        - this.db.select_hash — получение единственной записи
        - this.db.select_loop — проход по выборке
    - Модификация данных
        - this.db.do — исполнение произвольного SQL

Теперь расскажем о них подробнее.

Общие слова

Начнём с двух банальных утверждений:

- реляционная модель вообще, 3½-я нормальная форма Бойса-Кодда в частности и язык SQL играют в области программирования баз данных такую же роль как колесо, в частности, круглое колесо и, наконец, круглое металлическое колесо в транспортной отрасли;
- писать SQL вручную на каждый конкретный случай -- крайне неэффективно экономически.

Для тех, кто считает ООП столбовой дорогой в IT, стандартный ход мысли -- полностью подменить на уровне приложения реляционную модель объектной (в смысле жёсткой иерархии классов) и транслировать действия над объектами в SQL-запросы при помощи того или иного средства класса O.R.M.

Не будем углубляться в критику этого подхода, а просто предложим альтернативу, реализованную в Dia.js:

- покрыть ~85% используемых запросов лёгкими генераторами SQL из JSON-структур, без жёсткой типизации, но с использованием знаний о модели БД;
- оставить разработчику полный доступ к родному для используемой БД диалекту SQL.

this.db.add

По ходу формирования AJAX- и прочих RESTful-ответов нередко возникает потребность приписать в результирующий объект выборку, связанную с некоторой таблицей, под именем самой этой таблицы, по схеме:

```
data.my_table = // SELECT * FROM my_table
```

Для того, чтобы не дублировать идентификатор, можно использовать `this.db.add`: аналог this.db.list, который выдаёт в качестве своего значения не саму выборку, а результат её добавления в объект, заданный первым аргументом, с именем корневой таблицы, используемым в качестве ключа. Например, на выходе

```
        await this.db.add ({}, [

            {voc_user_options: filter},

            {'user_options(is_on)': {
                id_user: user.uuid,
            }}
        ])
```

будет объект вида

```
{voc_user_options: [
  {uuid: ..., "user_options.is_on": 1},
  {uuid: ..., "user_options.is_on": 0},
  ...
]}
```

this.db.add_all_cnt

Как this.db.add, данный метод приписывает результирующую выборку к заданному объекту под именем корневой таблицы.

Отличия состоят в том, что:

- выборка -- не полная, а усечённая (см. this.db.select_all_cnt);
- помимо списка записей, добавляется компонента `cnt` с полным числом записей (см. там же).

Пример:

```
return this.db.add_all_cnt ({}, [{users: filter}, 'roles AS role'])
```

Результат (после await):

```
{
  users: [{uuid: ..., label: 'Admin Adminov'}],
  cnt: 1
}
```

this.db.add_vocabularies

Добавление к заданному объекту одного или нескольких словарей данных: маленьких выборок, предназначенных для формирования HTML-элементов `SELECT` и подобных им деталей UI.

ВНИМАНИЕ: для таблиц, у которых в модели данных указано фиксированное содержимое, список берётся непосредственно из памяти, без обращения к БД.

Для остальных таблиц формируется запрос: по умолчанию на поля `id` и `label` с сортировкой по последнему, с возможностью переопределить это в опциях:

```
this.db.add_vocabularies (data, {
  roles: {},
  voc_orgs: {
    label  : 'label_full',
    filter : 'is_deleted=0',
    order  : 'ogrn',
  },
})
// data.roles    = [{id: 1, label: 'admin'}, ...]
  // SELECT id, label FROM roles ORDER BY label
// data.voc_orgs = [{uuid: '...', label: '...'}, ...]
  // SELECT id, label_full AS label FROM roles ORDER BY orgn
```

this.db.do

Исполняет SQL-запрос с заданным списком параметров.

```
await this.db.do (
  'DELETE FROM records WHERE id_type = ? AND year < ? ',
  [this.rq.data.id_type, 2019]
)
```

this.db.delete

Данный метод осуществляет удаление из указанной таблицы множества записей, удовлетворяющих заданному критерию.

Удаление — жёсткое, посредством SQL-оператора `DELETE`, а не установка каких-либо статусов, флагов и прочих полей данных.

Второй аргумент `this.db.delete` представляет собой js-объект, из которого формируется раздел `DELETE ... WHERE`. Он в точности соответствует набору фильтров [this.db.query]. В действительности SQL генерируется именно `this.db.query` — а потом перед `FROM` ставится `DELETE`.

В простейшем случае ключи — имена полей, а значения передаются как соответствующие параметры.

Если полученный SQL не предполагает ни одного параметра, вместо его запуска генерируется сообщение об ошибке: всё-таки осмысленный и необходимый `DELETE` на все записи в таблице представляется настолько гораздо более редким случаем, нежели следствие ошибки разработчика. Для полной зачистки таблицы всегда можно воспользоваться `this.db.do` с явно заданным SQL — и в большинстве таких случаев лучше подходит не `DELETE`, а `TRUNCATE`.

Пример:

```
return this.db [['delete', 'upsert'] [data.is_on]] ('user_options', {
  id_user:            this.user.uuid,
  id_voc_user_option: data.id_voc_user_option
})
```

this.db.delepsert

Данная функция осуществляет синхронизацию пакета данных, содержащего:

- имя таблицы;
- общую часть: js-объект ключей/значений, общих для всех рассматриваемых записей;
- список данных: массив js-объектов ключей/значений для остальных полей;
- ключ различения: имя поля, которое для каждой записи в списке имеет уникальное значение.

На основании этих данных формируется один или два вызова:

- всегда — this.db.delete для записей с указанной общей частью и значениями ключа, не встречающимися в списке данных;
- при непустом списке — this.db.upsert на его записи, куда добавлена общая часть.

Таким образом, после исполнения этого метода для заданной общей части

- указанные поля явно приведённых записей имеют требуемый вид
  - прочие поля (с точностью до действия триггеров)
    - для ранее существовавших записей имеют старые значения;
    - для вновь созданных записей — значения по умолчанию;
- прочих записей с такой общей частью быть не может.

Если первичный ключ таблицы составной, причём общая часть данных содержит все поля, кроме ключа различения — последний параметр можно не указывать.

Пример

Рассмотрим таблицу `user_users`, у которой всего 2 поля:

- `id_user` — ссылка на пользователя;
- `id_user_ref` — ссылка на выбранного им адресата.

Первичным ключом таблицы является комбинация `id_user` и `id_user_ref`.

Допустим, от имени текущего пользователя системы (`id_user=this.user.uuid`) требуется записать множество выбранных им адресатов `this.rq.data.ids` так, чтобы старые адресаты с не упомянутыми на этот раз номерами были бы удалены.

В данном случае:

- все записи объединяет их принадлежность к текущему пользователю — значит, общая часть имеет вид `{id_user: this.user.uuid}`;
- за вычетом поля `id_user` список записей формируется как `this.rq.data.ids.map (i => ({id_user_ref : i}))`
- ключом различения (идентификатором записей, уникальным внутри пакета) является `id_user_ref`;
  - поскольку он получается из первичного ключа таблицы выбрасыванием оттуда общей части — его можно не указывать.

Итак, описанная задача решается посредством вызова

```
return this.db.delepsert (
  'user_users',
  {
    id_user     : this.user.uuid
  },
  this.rq.data.ids.map (i => ({
    id_user_ref : i,
  })),
// 'id_user_ref', — можно не указывать
)
```

this.db.fold

Проход посредством this.db.select_loop по выборке согласно SQL и параметрам, вычисленным this.db.query

```
user.opt = await this.db.fold ([
  {'user_options()': {
      is_on: 1,
      id_user: user.uuid
  }},
  'voc_user_options(name)'
], (i, d) => {d [i ['voc_user_options.name']] = 1}, {})
```

this.db.get

Извлечение посредством this.db.select_hash первой записи согласно SQL и параметрам, вычисленным this.db.query

```
let data = await this.db.get ([
  {users: {uuid: this.rq.id}},
  'roles AS role'
])
```

this.db.insert

Данный метод предназначен для создания в заданной таблице новых записей, представленных "объектами" javaScript.

```
await this.db.insert ('users', {
  id:   this.rq.id,
  label: this.rq.data.label,
  // ... прочие поля
})
```

Если таблица генерирует значения своего первичного ключа и значение соответствующего поля не задано, ключ вновь созданной записи передаётся в качестве результата операции:

```
let id = await this.db.insert ('news', {label: 'Новая новость'})
```

Для сценариев, где, напротив, требуется либо создать запись с заранее известным ключом, либо не менять данные вовсе, предусмотрен специальный проверочный метод this.db.is_pk_violation

```
try {
  await this.db.insert ('payments', payment)
}
catch (x) {
  if (this.db.is_pk_violation (x)) return payment; else throw x
}
```

Если второй аргумент является массивом, в результате вызова может быть создано множество записей:

```
await this.db.insert ('children', [
  {parent, ord: 1, label: 'one'},
  {parent, ord: 2, label: 'two'},
])
```

this.db.insert_if_absent

Данный метод вставляет указанную запись в таблицу -- но только в том случае, если там нет содержимого с таким же первичным ключом.

Если же данный ключ был добавлен ранее, никаких изменений не производится.

```
////////////////////////////////////////////////////////////////////////////
do_create__default:
  async function () {
    let data = this.rq.data
    await this.db.insert_if_absent (this.module.table, data)
    return data
 },
```

Соответственно:

- в отличие от this.db.insert:
  - указание первичного ключа обязательно (иначе -- ошибка);
  - результат -- всегда пустой;
- в отличие от this.db.upsert:
  - если содержимое неключевых полей отличается от переданного, оно остаётся таким же, а переданные значения игнорируются.

Последнее не связано с риском потери данных, поскольку предполагается, что:

- INSERT для заданного ключа может быть только один;
- если данные в базе есть -- они не менее, а, возможно, более свежие, чем содержимое передаваемой записи.

Вообще данный метод предназначен для сценариев регистрации внешних запросов, принимаемых с риском повтора. В ситуации, когда сторонняя система N+1-й раз отправляет одну и ту же заявку, потому что не запомнила успешный ответ предыдущие N раз (из-за проблем с каналом связи или собственных ошибок), она должна получить не отказ из-за нарушения уникальности, а сообщение об успехе -- как будто её заявка зарегистрирована только что.

this.db.is_pk_violation

Вместо этого метода рекомендуется использовать this.db.insert_if_absent.

this.db.list

Извлечение посредством this.db.select_all выборки согласно SQL и параметрам, вычисленным this.db.query

```
user.peers = await this.db.list ([
  {'users(uuid, label, uuid AS id)': {
    'login<>' : null,
    'uuid <>' : user.uuid,
  }},
  {'$user_users ON user_users.id_user_ref = users.uuid' : {
    id_user : user.uuid,
    is_on   : 1,
  }}
])
```

this.db.load

Данный метод осуществляет загрузку строк из предоставленного потока в указанную таблицу оператором `COPY ... FROM STDIN`.

Для его использования необходимо включить в проект модуль pg-copy-streams.

```
try {
  await this.db.load (
    fs.createReadStream (fn),
    'my_table',
    ['id', 'label'],
    {ENCODING: 'utf-8', NULL: ''}
  )
}
catch (x) {
  darn (x)
}
```

Последний параметр представляет собой набор опций для формирования раздела `WITH`.

Если он не задан, по умолчанию принимается значение `{NULL: ''}`, то есть текстовый формат с разделителями-символами табуляции и переводом пустых строк в `NULL`.

this.db.select_all

Извлекает список объектов, соответствующий выборке по SQL-запросу с заданными параметрами

```
let users = await this.db.select_all (
  "SELECT * FROM users WHERE is_deleted = 0 AND id_role = ? ORDER BY label", [
    this.rq.data.id_role,
  ]
)
```

this.db.select_all_cnt

Комбинированный результат запуска двух вызовов:

- this.db.select_all на исходный SQL с ограничением величины выборки;
- this.db.select_scalar на соответствующий `SELECT COUNT (*)`.

```
let [all, cnt] = await this.db.select_all_cnt (
  'SELECT * from users WHERE id_dep=? ORDER BY label'
  , [id_dep]
  , 50            // limit
  , 0             // offset
)
```

this.db.select_hash

Извлечение первой записи заданной выборки.

```
let user = await this.db.select_hash ('SELECT * FROM users WHERE id = ?', [id])
```

this.db.select_loop.asciidoc

this.db.select_scalar

Извлечение единственного поля первой записи из заданной выборки.

```
let v = await this.db.select_scalar (
  'SELECT value FROM settings WHERE key = ?',
  [this.rq.id]
)
```

this.db.update

Данный метод предназначен для изменения содержимого записей, заранее созданных в БД.

В простейшем варианте он обновляет запись с заданным значением первичного ключа:

```
await this.db.update ('tasks', {
  id: 1,
  id_status: 30,
}) // UPDATE tasks SET id_status = 30 WHERE id = 1
```

Впрочем, набор полей для выражения WHERE можно задать явно:

```
await this.db.update ('orgs', {
  inn: '11111111111',
  kpp: '111111111',
  label: 'ЗАО "ВЕКТОР"',
}, ['inn', 'kpp'])
```

Этот ключ не обязан идентифицировать запись однозначно. Массовые обновления оформляются аналогично:

```
await this.db.update ('employees', {
  id_dep: 13,
  dt_fired: new Date (),
}, ['id_dep'])
```

Как и this.db.insert, this.db.update предполагает пакетный режим исполнения:

```
await this.db.update ('users', [
  {login: 'admin', id_role: 1},
  {login: 'user', id_role: 2},
], ['login'])
```

this.db.upsert

Данный метод гарантирует, что после его исполнения в указанной таблице будет присутствовать запись с указанными значениями полей — вне зависимости от того, была она создана ранее или появится только сейчас.

Например, вызов

```
await this.db.upsert (
  'user_options',
  {
    id_user:       this.user.id,
    id_voc_option: this.rq.data.id_voc_option,
    is_on:         1,
  },
  [
    'id_user',
    'id_voc_option',
  ]
)
```

приведёт к тому, что в таблице `user_options` у записи с указанными значениями ключевых полей `id_user` и `id_voc_option` в поле `is_on` будет значение `1`. Возможны 3 ситуации:

- до вызова такой записи не было — тогда она будет создана;
- до вызова была запись с иным значением `is_on` — тогда она будет обновлена;
- до вызова запись уже соответствовала требованиям — тогда данные не изменятся.

Программисту не надо прописывать 3 ветви алгоритма — `this.db.upsert` реализует их все.

Однако для того, чтобы использовать этот метод, необходимо предусмотреть композитный первичный ключ либо UNIQUE-индекс по набору ключевых полей: в данном случае это `id_user` и `id_voc_option`.

Если 3-й параметр -- набор ключевых полей -- совпадает с первичным ключом таблицы, его можно не указывать:

```
this.db.upsert ('user_users', ids.map (i => ({
  id_user     : this.user.uuid,
  id_user_ref : i,
})))
```

this.this.http — HTTP клиент

Аналогично this.db, `this.http` — не фиксированное имя переменной (в отличие от this.user и ей подобных), а типовое обозначение ресурса определённой природы: HTTP-клиента. Точнее, тонкой обёртки над родным HTTP-клиентом node.js: http.request ().

Класс HTTP предназначен для хранения пакета параметров, переиспользуемых от вызова к вызову `http.request ()`. Во избежание разночтений, отметим: он не реализует функциональности хранения соединений (connection pooling): эта задача успешно решена стандартным http.Agent.

Что же, собственно, делает объект, позиционируемый как HTTP-клиент в Dia.js? Прежде всего, он хранит переданные при инициализации опции запроса -- поддерживаемые `http.request ()` и дополнительно ещё опцию `url`:

```
this.pools = {
  //...
    http_some_soap_svc: new HTTP ({
      url: this.ws.some_soap_svc.url,
      timeout: 100,            // например
      agent: http.globalAgent, // по умолчанию
      //...
    }),
    http_my_rest_svc: new HTTP ({
      url: this.ws.my_rest_svc.url,
      //...
    }),
```

```
  //...
}
```

Если задан `url`, он определяет `protocol`, `hostname`, `port` и `path`. Любую из этих и прочих опций можно переопределять при выполнении запроса. Предполагается, что в основном это должен быть только `path`.

Весь API нашего клиента сводится к единственному асинхронному методу `response`, который либо возвращает тело HTTP запроса, либо генерирует исключение: в том числе для ответов с кодом, отличным от `200 OK`.

```
let rp = await this.http_my_rest_svc.response (
  {path: "/1.1/rest/resource/somePath"},       // дополнительные опции
  this.rq.data                                 // тело запроса
)
let rp = await this.http_my_rest_svc.response (
  {path: "/1.1/rest/resource/otherPath"},      // ... а тела нет
)
let soap_response = await this.http_some_soap_svc.response (
  {},                                          // все опции -- в настройках
  this.rq.data                                 // тело запроса
)
```

Как видно из приведённых примеров, описываемый клиент предназначен для вызовов SOAP- и RESTful-сервисов с ограниченными объёмами запросов и ответов. Для них применяются следующие эвристики, подходящие в большинстве практических случаев:

- если тело запроса не задано, `method` определяется как `GET`;
- в противном случае -- как `POST`, причём
  - если не заданы `headers`, то выставляется единственный заголовок `Content-Type`, значение которого определяется по первому символу тела:
    - `text/xml` для <;
    - `application/json` для `{` или `[`.

Любую из упомянутых опций для конкретного запроса можно переопределить явно.

На момент написания этого текста тела запросов и ответов предполагаются строками, а кодировка -- `utf-8` при передаче в обоих направлениях.

this.this.timer — таймер

Аналогично this.db, `this.timer` — не фиксированное имя переменной (в отличие от this.user и ей подобных), а типовое обозначение ресурса определённой природы: таймера, предназначенного в основном для обработки хранимых очередей отложенных запросов.

Таймер -- это объект, инициирующий асинхронный запрос `todo` фиксированного вида:

- как можно ближе к назначенному времени;
- но не ранее, чем завершится прошлый запрос, инициированный этим таймером;
- и не ранее чем пройдёт `period` мс с начала прошлого запуска.

Конфигурация

Таймер задаётся в конфигурации приложения тремя параметрами:

```
this.pools = {
  //...
  erp_rp_timer : new Timer ({
    label: 'ERP responses queue', // для log-файла
    period: 5000,                 // мс между запусками
    max_locks: 5,                 // ограничение на блоки (см. ниже)
    todo: [Async, {
      rq: {type: 'organization_imports', action: 'check'},
      user:  {uuid: '00000000-0000-0000-0000-000000000000'},
      conf:  this,
      pools: {db: this.db2}
    }]
  }),
  //...
}
```

Таймер устанавливает в создаваемых обработчиках ссылку на самого себя и становится виден там как `this.timer`.

Отметим, что параметр `period` означает лишь ограничение на частоту запусков: не более раза в `period` мс. По умолчанию его значение равно `0`.

Так или иначе, сам по себе таймер срабатывать не будет. Его необходимо явно установить:

- либо на старте приложения (при загрузке конфигурации):

```
this.pools.erp_rp_timer.on () // здесь this -- это конфигурация
```

- либо при обработке запроса (в том числе инициированного самим таймером):

```
this.erp_rp_timer.in (1000)   // ... а здесь this -- это обработчик
```

Запуск

Метод `in (ms)` -- основной способ установки таймера, его параметр -- число миллисекунд, до момента, ранее которого вызывать процедуру не имеет смысла.

Чтобы запустить `todo ()` в ближайший допустимый момент, можно вызвать `in (0)` или, что то же самое, `on ()` без параметров.

Если в момент вызова `in (ms)` таймер установлен на некоторый момент в будущем, то он либо переустанавливается на новое время (если оно ближе, чем ранее запрошенное), либо остаётся без изменения.

Если в момент вызова `in (ms)` таймер занят выполнением `todo ()`, то его установка откладывается на момент после завершения процедуры.

Если назначенный таймеру момент наступает, пока не окончен прошлый запуск `todo ()`, то очередной запуск будет отложен до окончания предыдущего.

В любом случае:

- после каждой установки таймера (вызова `in (ms)` или `on ()`) процедура `todo ()` будет вызвана;
  - при этом на много параллельных `in (ms)` может приходиться один вызов `todo ()`;
- вызовы процедуры `todo ()` из-под таймера не будут пересекаться во времени и не будут происходить чаще, чем раз в `period` мс.

lock / max_locks: управление пропускной способностью

Описываемый таймер в основном предназначен для обслуживания очередей отложенных операций. Соответственно, инициируемый им запрос (типовое имя действия: `check`) должен проверять наличие заявок и запускать их обработку.

Таймер обеспечивает то, что сама процедура `check` будет вызываться только в однозадачном режиме, причём не слишком часто. Однако из-под `check` могут запускаться параллельные процессы обработки — и их количеством необходимо управлять. Для этого предусмотрена опция `max_locks` и метод `lock`.

Допустим, процедура проверки очереди извлекла массив уникальных имён объектов, требующих обработки. Назовём его `files`. Если этот список пуст, делать нечего:

```
if (files.length == 0) return darn (this.uuid + ": the queue is empty, nothing to do")
```

В противном может оказаться, что некоторые его элементы уже находятся в обработке. Чтобы не порождать дублирующие процессы, можно запросить список объектов блокировки:

```
let locks = this.timer.lock (files)
```

На выходе будет список ресурсов, соответствующий нетронутым объектам, причём длины не более `max_locks` за вычетом числа запросов в работе. Если он окажется пустым — это значит, что в текущий момент идёт `max_locks` параллельных процессов и новые порождать нельзя:

```
if (locks.length == 0) darn (this.uuid + ": all workers busy")
```

В противном случае для каждого полученного блока можно запустить свой обработчик. Исходная строка-идентификатор вычисляется как поле `lock.key`:

```
    for (let lock of locks) {
        new (require ('./Handler/File')) ({
            user:  {uuid: '00000000-0000-0000-0000-000000000000'},
            conf:  this.conf,
            rq:    {
                type: this.rq.type,
                action: 'process_file',
                path: lock.key,
            },
            pools: {
                db: this.conf.pools.db,
                timer: this.timer,
                lock // ВНИМАНИЕ! Это важно
            }
        }).run ()
    }
```

Обратим внимание на то, что `lock` передаётся новому обработчику в качестве ресурса. В результате он освобождается в последней фазе запроса, наряду с `db` — таким образом таймер узнаёт о его завершении.