# Цель

Общая идея состоит в том, чтобы разработать комплекс библиотек, с помощью которого потом можно было бы быстро и легко создавать веб-приложения (ajax) под любые нужды и задачи. Здесь изложены основные идеи и концепции, которые предполагаются в системе.

В дальнейшем получать прибыль от создания таких сайтов непосредственно, либо через специальный генератор сайтов (веб-сервис), либо продавая лицензию на использование движка/библиотеки третьим фирмам.

# Функции

## Переключение шаблонов/языков «на лету»

За счёт быстрой замены css-стилей. Для этого, правда, придётся заняться css-вёрсткой дизайнов, оставив в html только сам контент и весьма базовую разметку. И уж тем более никаких разметочных таблиц. Зато контент будет оптимален для поисковой оптимизации.

## Удобное управление раскладкой интерфейса

Админ должен в WYSIWYG-режиме создать начальный расклад модулей по странице, а юзер должен иметь возможность поменять это под свои предпочтения.

Но при этом не должно всё сводиться к модульности клиентской части, а кое-что по желанию админа должно быть зафиксировано и являться частью статического оформления.

## Интерактивная подсказка при заполнении форм

Когда заполняются поля, которые имеют типовые значения, то приложение обращается к серверу за списком предложений и выводит его для выбора. Всё «на лету».

## Редактор контента прямо при просмотре

Нажимаешь кнопочку —и просмотр превращается в редактор. И наоборот.

Но этот режим противоречит принципу чёткого разделения админской и юзерской части (управление и смотрение), поскольку редактор — это уже админская часть.

Как компромисс можно сделать так, чтобы в админской части был редактор, и оно задействовало юзерскую часть для очень подробного превью (примерно как Print Preview в MS Word).

Либо, ещё как вариант, делать две разные админские части: просто редактор (может быть даже и WYSIWYG-like), и inline-редактор, который показывает контент как он есть, и позволяет его тут же редактировать. В этот же редактор включить как функциональную часть редактор раскладки контента (то есть drag-and-drop раскладка текстовых блоков, картинок и прочего контента).

## Модульные WYSIWYG-редакторы

Возможность подключения сторонних WYSIWYG-редакторов либо напрямую, либо через мелкий wrapper. И чтобы пользователь мог выбирать чем ему пользоваться (как в настройках, так и непосредственно в момент редактирования с быстрой заменой редактора «на лету»).

## Умная вставка в WYSIWYG-редактор

То есть определяет что в редактор вставлен какой-то типовой известный тэг (ссылка на картинку, на youtube и т.п.), и автоматически вставляет его в код как соответствующий элемент, а не как текст тегов.

## Легкая и быстрая установка модулей

Модули чтоб можно было и загружать как файлы, и выбирать с официального сервера модулей (запрос делает серверная часть).

Удалять и приостанавливать модули тоже должно быть легко.

## Модификация функционала без модификации дистрибутивного кода

То есть способность дописывать новый функционал так, чтобы при апдейте системы его не затронуло, а он работал бы и дальше. Это включает как расположение файлов, так и способ подключения такого функционала (и серверное, и клиентское).

## Чёткое разделение данных и оформления

Данные строго в XML, оформление строго в XSL/XSLT(+CSS?).

## Пакетные запросы

Клиентский компонент хранит у себя очередь того, что надо передать на сервер, причём джля каждой передачи указывается её срочность и важность и зависимость от других передач, а также режим блокирования до получения результата или же фоновый режим. Он передаёт это всё на сервер одним-единственным запросом либо по интервалу времени, либо по появлению «срочного» запроса. Возвращаемые данные передаёт в callback-процедуру, указанную для каждого запроса, или возвращает как результат для «срочного» запроса.

## Автоповтор запросов при сбое

Включая как запросы на данные (GET-like), запросы на выполнение действия (POST-like), так и загрузки файла. То есть если загрузка файла сбойнула, то она будет повторена.

## Отслеживания передачи запроса

Все пакетные запросы создают события в интерфейсе, которое показывает что сейчас делается (в какой-нибудь строке статуса). Это сообщение обновляется регулярно (по таймеру или по событию). Там же отображается, какой повтор делается, либо же это первый запрос.

## Отслеживание аплоада в процентах

Использование perl-скрипта для отслеживания прогресса загрузки файла на сервер. Пока это единственное известное решение для подобных целей.

## Многопотоковый либо последовательный аплоад

## Поисковая оптимизация страниц

Для поисковых движков все адреса должны быть сведены к простой форме (без знака «?» и лишних параметров). А ещё лучше если все адреса будут управляемые (то есть админ задаёт структуру адреса, а система уже под неё подстраивается).

Также не забыть про оптимизацию ключевых слов, описаний и прочих мета-тегов для каждой страницы и каждого контентного модуля (для автоматического складывания списка ключевых слов и title-заголовка).

## Поиск по контенту сайта

Такая система, которая сама всё что надо индексирует и потом выводит при поиске. В том числе её возможное применение — автоматическое нахождение «схожих» статей по ключевым словам. Ещё одно применение — интерактивный поиск по сайту.

В том числе поиск по картинкам (их описательной текстовой части и ключевым словам) и прочим нетекстовым ресурсам.

## Транзакционность операций

Все производимые операции не применяются к готовому сайту до тех пор, пока не будет нажата кнопка применения (аналог commit).

Рассмотреть возможность сохранения изменения на стороне сервера во временном режиме (то есть изменения сохранены, но фактически не применены); чтобы потом по списку незавершённых операций можно было вернуться и продолжить их (завершив или отменив в конечном итоге). В простейшем, но не самом лучшем случае — это checkin/checkout. Но желательно всё-таки рассмотреть вопрос задействования какой-либо версионной системы (svn предпочтительно, либо же самодельной, что сложно).

Транзакционность особенно актуальная для загружаемых файлов, так как загрузить их можно лишь единожды, а потом лишь на них ссылаться по идентификатору, по которому серверная часть найдёт их где-то во временном серверном хранилище.

## Чёткое разделение админской и юзерской частей (?)

То есть есть модули админские для управления, а есть юзерские для просмотра.

Правда, тогда неясно как делать «редактирование на лету» — ведь это явное совмещение функционала в один модуль.

В общем, этот пункт ещё ПОД ВОПРОСОМ. Но смотри пункт «редактор контента прямо при просмотре» — там есть возможное решение (с тремя модулями: юзерский просмотр, админский простой редактор, и админский inline-редактор).

## Гибкое управление полномочиями пользователей

Достигается с помощью цепочки «пользователь—группа—полномочие»; при этом в коде страниц и модулей вшито или задано в конфиге исключительно имя полномочия, а группы предназначены лишь для пакетного управления полномочиями по пользователям.

## Легкая инсталляция

Просто залить файл дистрибутива, распаковать его и открыть специальный адрес инсталляции. Дальше поможет веб-визард.

Либо ещё лучше: один-единственный файл php, и в нём вшит архив. Заливаешь его, запускаешь, и оно само спрашивает куда и как распаковаться и запускает визард.

Или ещё как вариант: ставишь мелкий скрипт инсталлятора, запускешь его, а он предлагает скачать дистрибутив и нужные модули с официального сайта (запросы делаются с серверной стороны).

## Пакетное управление модулями

Все модули распространяется пакетами (модульпаками). Устанавливается именно пакет. В пакет входят: файлы с классами модулей, сколько бы их там ни было (причём для каждого указывается адрес, откуда его обновлять), сценарии внедрения пакета в БД, сценарии удаления пакета из БД, и т.п.

Поставленный пакет ничего в функционал не вносит, и лишь делает доступными классы для модулей, а также в менеджере пакетов отслеживает обновления этого пакета.

Уже на основе имеющегося пакета можно создать рабочие модули, каждый привязан к своему классу, имеет ряд зарегистрированных событий и т.п. Причём дефолтные наборы событий, дефолтные конфиги берутся из информации в пакете. Также вызывается скрипт интеграции в базу данных, который учитывает все префиксы/имена полей и таблиц, указанные для этого модуля в его конфигах.

Установленный и включенный модуль, собственно, и начинает работать.

При установке пакета, после самой установки как раз и вызывается мастер создания модуля на основе этого пакета. При удалении пакета, аналогично, вызывается мастер удаления всех модулей, построенных на основе этого пакета.

## Всё в уникоде

Как наиболее вероятный вариант – передачи контента в UTF8. Хранение даны в базе, скорее всего, в UCS2.

Но желательно чтобы можно было настраивать. В том числе даже настраивать кодировку выводимого контента (преобразуется из уникода в неё на выводе, и из неё в уникод на вводе).

## Кэширование элементов страниц и самих страниц

Как внутриядерное кэширование для сокращения количества запросов к базе, так и оптимальная работа с http-заголовками для оптимизации под кэширующие прокси и кэши браузеров.

## Полуавтоматическое обновление движка

Само обращаетсяна официальный сайт и качает апдейты. При этом не страдает переделанный функционал система, уже установленные сторонние модули и т.п.

Либо обновление по крону.

## Внутренний cron-менеджер

Чтобы не заморачиваться с механизмами реализации крона на каждом хостере, сделать в системе свой cron. А на хостере сделать лишь одну-единственную запись для активирования встроенного крона каждую минуту (ну или чуть реже).

В крон вписывать различные задачи. Например, обновление системы, рассылку почты из очереди, генерацию статистик и отчётов, очистку временных файловых хранилищ от старого неиспользованного файлья, создание резервных копий (контента), и т.п.

# Архитектура

## Общая

Система состоит из двух частей: клиентской и серверной.

## Серверная сторона

### Обзорно

Серверная часть представляет собой скрипты на языке php, работающие на сервере хостера, и принимающие данные от клиентской стороны и отдающие ей данные.

Задачи серверной стороны: обработка данных и формирование вывода, проверка прав доступа, ведение сессий работы, аутентификация и авторизация пользователя.

Серверная часть построена как набор модулей, взаимодействующих друг с другом через ядро с помощью событий.

### Модульная структура

Каждый модуль реализован как самостоятельный класс серверного языка программирования и хранится в отдельном файле с тем же именем, что и имя класса, и с соответствующим расширением, в специальной папке с модулями. Классы могут выстраиваться в иерархии, если требуется; могут быть абстрактными, если они являются лишь заготовкой для своих наследников. Для модулей пригодны лишь классы, от которых можно создать экземпляр. Желательно чтобы все модули происходили от одного предка; но обязательным это не является, достаточно лишь чтобы они реализовывали минимально необходимые методы с правильным набором параметров (в основном, конструктор).

В конфиге сайта указывается какие модули создавать, причём каждый такой модуль в обязательном порядке характеризуется своим кодовым именем и именем класса, реализующего функционал модуля; опционально для модуля указывается БД-модуль, обслуживающий подключение этого модуля к базе данных (если оно отлично от того, что использует ядро и сайт в целом). Модули не создаются (и их классы не читаются из файлов) сразу при запуске ядра, а лишь при их вызове для реакции на событие.

### Событийное взаимодействие

Любой модуль в любой момент своего функционирования может вызвать любое нужное ему событие. Для события в обязательном порядке указывается его кодовое имя (которое, как правило, жёстко вшито и изменению не подлежит), тип события (запросное ли оно, или же оповестительное); необязательно указывается один аргумент данных (который может быть массивом или хешем). Генерация события происходит вызовом соответствующего метода ядра.

В конфиге сайта кроме списка модулей хранится и подписка модулей на события, в которой указывается: кодовое имя ожидаемого события, модуль, который нужно вызвать для реакции на событие, метод этого модуля (если не указан, то будет вызван метод dispatch), приоритет (порядковый номер) этого обработчика события, карта отображения ключей в параметре с данными (только если оно хеш), и прочие вспомогательные детальки.

При возникновении события ядро вызывает все подписанные на это событие модули и методы в порядке возрастания приоритета, не забывая при этом производить отображение ключей хеша. Если при вызове события было указано, что это запросное событие и может возвращать результат, то первый не-null результат и будет возвращён как результат события. Если же событие было указано как оповестительное, то результаты вызова обработчиков игнорируются. Так или иначе, любой обработчик может прекратить дальнейшую обработку события генерация исключения «abort\_event»; для события-запроса в качестве результата при этом будет возвращено null.

Все взаимодействия модулей сводятся исключительно к генерации и обработке событий; иным другим способом модули взаимодействовать не должны. Таким образом, появляется возможность подмены одних модулей другими с помощью просто замены обработчиков событий.

Для «начального толчка» ядро генерирует оповестительное событие для каждой стадии своей работы. Реакции модулей на эти события и формируют функционал сайта.

## Клиентская сторона

### Обзорно

Клиентская часть представляет собой библиотеку на языке javascript, нормально работающую на большинстве или на всех браузерах.

Задачи клиентской стороны: отображение данных, интерактивный интерфейс, реакция на операции пользователя, загрузка и выгрузка данных с серверной стороны.

Клиентская часть реализована как набор классов, подгружаемых браузеров и выполняемых на стороне клиента (посетителя).

## AJAX-взаимодействие

### Обзорно

Клиентская часть должна интенсивно взаимодействовать с серверной частью для обмена данными. При этом обязательно пользователю должно показываться что идёт взаимодействие, а желательно указывать что это за взаимодействие и на каком оно сейчас этапе.

С протоколом взаимодействия пока не определился, но это либо JSON, либо XML. Или, что скорее всего, выбор протокола ляжет на плечи разработчика сайта; с моей стороны нужно будет лишь предоставить оба способа.

### Пакетные запросы

### Серверная обработка

Поскольку обработка мелких данных при AJAX-взаимодействии отличается от обработки страниц целиком, то ядро должно как-то по-особому реагировать на такие мелкие, но частые запросы. Причём это должна быть забота именно ядра или околоядерных модулей, так как AJAX в данном случае позиционируется как основной элемент системы.

## Резервная система

Клиентская сторона может оказаться отключенной у пользователя по тем или иным причинам. Например, браузер не поддерживает javascript вообще, либо он отключен в настройках безопасности. В таком случае общий комплекс библиотек должен работать таким образом, чтобы пользователь мог работать с системой по старому принципу «запрос-ответ».

Вероятно, в самом простом варианте, такая система является просто надстройкой над серверной частью, которая на серверной стороне производит все запросы и обрабатывает вывод от серверной части, и отдаёт готовый ответ пользователю.

Но так или иначе, клиентская часть должна быть построена таким образом, что в случае отсутствия javascript, сама по себе задействовалась бы резервная система. То есть ссылки по умолчанию должны быть настроены на эту резервную систему; а также все структуры данных и т.п.

## Двухсторонние компоненты

Все компоненты должны быть потенциально двухсторонними: поддерживать своё состояние (данные) на серверной и на клиентской стороне.

В частности, это полезно когда открывается ссылка с выключенным яваскриптом или в новом окне (события при этом игнорируются). Система в целом должна сработать таким образом, чтобы воспроизвести новую страницу на основе исходной, в которой определённый компонент изменит своё состояние.

Для этого компонент должен хранить своё состояние на сервере, дуплицируя его на клиенте. На клиенте же хранятся и все вспомогательные временные данные.

Система резервирования в таком случае как раз и делается через эту двухстороннюю связь компонентов.

# Реализация

## Иерархия клиентских классов JS

Весь клиентский код должен быть организован в классы языка JS.

Предполагаемые классы визуальных элементов: DragArea (место, по которому можно таскать элементы), DragItem (то, что можно таскать по DragArea), Numeric (поле, в которое можно вводить только числа), и т.п. Для соединениея с элементом html используют метод attach(id). При необходимости есть метод attachNewDiv() и т.п., которые создают тег на лету.

Предполагаемые статические классы ядерных утилит: Timer (для для задания таймеров), Request (для запроса-ответа к серверной части) и все его потомки для разных типовых структур данных. Скорее все эти классы имеют по одиному-единственному экземпляру, которые упрятаны в экземпляр ядра по соответствующим полям (хотя еще вопрос, а надо ли оно там, или оставил их глобальными переменными).

Каждый класс хранится в своём отдельном файле, имя которого содержит имя класса и расширение «.js». Все классы лежат в одной папке; либо, как вариант, в иерархии папок, а при загрузке класса путь к папке используется лишь как подсказка.

Для загрузки класса используется конструкция типа «ClassLoader->require(“hint/path/classname”);», где ClassLoader — это предопределенный экземпляр статического класса. Этот класс кеширует какие классы уже загружены, и повторную загрузку не производит. Эти конструкции должны быть также указаны и в каждом файле каждого класса, если он зависит от других классов.

Если js-файл запрошен напрямую, то он выполнится и автоматически подгрузит всю необходимую ему иерархию классов.

Кроме того, должен быть серверный скрипт, который получает в аргументах список необходимых классов, читает их файлы, а также все файлы зависимостей по этой указанной выше конструкции (элемент серверного парсинга файлов), склеивает их в один единственный файл и отдаёт в ответ. Этот скрипт можно сделать и с помощью псевдо-урла, то есть если сам скрипт лежит в «http://server/path/js/index.php», то классы можно запросить по ссылке «http://server/path/js/class1+class2+class3». Такие ссылки весьма успешно будут кешироваться, и повторная загрузка не потребуется.

Надо только придумать как НЕ загружать классы, которые могут быть загружены этим скриптом по зависимостям, но уже загружены в клиента. Чтобы не было попыток повторной загрузки. Например, ссылками типа «http://server/path/js/class1+class2+class3-alreadyclassA-alreadyclassB». А поскольку уже загруженных классов может быть много, то и урл может получиться очень длинным.

А вот если отказаться от кеширования этих файлов браузером, то можно передавать списки нужных и имеющихся классов в POST-запросе. Тогда ограничение на длину запроса не будет проблемой. Проблемой станет загрузка классов при каждом запросе страниц. Но поскольку такие ajax-страницы загружаются лишь раз, а потом просто работают, то проблема эта средней тяжести.

С другой стороны, можно совместить оба способа, и загружать классы заранее в одном-единственном <script>-тэге, который ссылается на страничку по такому урлу. Причём делать это ДО того, как какой-либо код начнёт выполнение на странице и сможет подгрузить другие классы.

## Типы запросов к серверу

Явно выделяются такие типы запросов к серверной части:

JS-склеиватель классов. Получает список необходимых классов, список уже имеющихся, загружает файлы этих классов и все зависимости, склеивает воедино, и отдаёт клиенту как есть, без пост-обработки, в обычно текстовом формате.

AJAX-данные. Получает запрос на какие-то данные. Проводит проверку прав на эти данные. Возвращает либо закодированные данные (JSON/XML), либо код и сообщение об ошибке.

AJAX-приложение. Основная страница, которую запрашивает сами клиент по урлу, и которая уже потом подгружает данные. Кроме того, сама по себе устроена как обычная страница с полями и ссылками, обращающимися к «резервной системе» если JS отсутствует.

Резервные страницы(?). Страницы, на которые ведут дефолтные урлы в страничке AJAX-приложения. Передаёт данные соответствующему AJAX-модулю, получает от него ответ, вклеивает его в общий шаблон страницы как положено (эмулируя само AJAX-приложение), и возвращает результат.

## СУБД и БД-модули

Вопрос с выбором СУБД весьма интересен.

С одной стороны, мы уже чётко определили некоторые требования к серверу (язык php) и клиенту (javascript, html). Поэтому имеет смысл также чётко определить и СУБД (mysql).

С другой стороны, если у нас есть возможность где-то предоставить выбор потребителю, то так и стоит сделать. К тому же, например, требования к веб-серверу не предъявляются и отдаются на откуп админа (apache или IIS или ещё чего).

К тому же не следует забывать, что ещё в задумке была потенциальная многобазовость сайта (то есть когда одни модули работают с одной базы, другие — с другой). И эта многобазовость может быть сделана вообще и на разных типах СУБД.

И при этом производится чёткое разделение на сам функционал модуля и на систему хранения данных (которая выбирается в зависимости от настроек модуля и его БД). Но это было там, где у нас было шаблонирование средствами PHP. Сейчас же речь идёт о генерации подходящего XML, и только об этом.

Кроме того, основной функционал включает в себя: проверку прав на операцию, предобработка данных из запроса, пост-обработка данных для ответа (включая генерацию DOM XML), протоколирование действий.

В общем, хотя потенциальная гибкость модульной системы и заманчива, она своим существованием усложняет жизнь как мне (например, как обновлять модули и нумеровать версии при изменениях в их БД-части), так и разработчикам сайтов (слишком сложно для понимания, равноправные модули проще). Поэтому, от системы БД-привязок к основным модулям отказываемся, и делаем простую плоскую модульную систему: 1 класс = 1 файл = 1 модуль. Разные реализации одного модуля под разные СУБД делаются как отдельные модули со схожим (и повторяемым) кодом, но всё-таки абсолютно отдельные. Весь код — и основной функционал, и генерация запросов, и вызовы API СУБД — пихается в один модуль-класс. Основная разработка ведётся по MySQL. Остальное как получится, либо просто неофициально.

Поскольку в событийной модели серверной части ядра у нас есть специальный тип событий для запроса данных, можно рассмотреть следующий механизм разделения модулей на разные СУБД. В основной модуль пишется основной функционал системы; из него шлются события-запросы на получение данных. В вспомогательных модулях эти события ловятся, данные получаются и возвращаются; эти модули сделаны каждый для своей СУБД. При всём при этом соблюдается принцип плоской модульной структуры, а также становится возможным создавать разные БД-реализации для одного и того же основного модуля. Недостатком при этом является необходимость пары модулей в большинстве случаев (основной и БД), но этот недостаток будет терпим, если будет приличная умная система управления модулями и их организацией по пакетам (модульпакам). Зато ощутимый плюс — куда как более простой код ядра (отсутствует управление БД-подмодулями), и вызванная этим чуть большая эффективность и быстродействие кода.

Так и сделаем, как в предыдущем параграфе.

## Модульная обработка результатов

Разные модули в зависимости от запроса возвращают разные данные и даже типы данных. Ими могут быть:

* XML-данные без оформления
* HTML-данные с оформлением
* XML-данные для AJAX (пакетный ответ)
* JSON-данные для AJAX (пакетный ответ)
* CSS&JS сгенерированные данные
* Текстовые данные для писем (в т.ч. для мейлеров)
* Бинарные данные (картинки, файлы)
* Multipart-данные из всех перечисленных типов
* Протокольные данные о выполнении для cron-скриптов

Таким образом, строгое ориентирование ядра на накопление XML-данных и последующая их обработка с помощью XSLT не оправдана. Имеет смысл разнести накопление и обработку контента по модулям, каждый из которых обрабатывает свой тип контента, и его выводит.

Для этого в ядре делаем метод void content (string type, mixed data) для добавления контента определённого типа, причём тип задаётся не какой-то описательной строковой константой, а именем модуля, в которые эти данные надо закинуть. Также потребуется метод получения от накопителя результата каких-то значений (например, DOMDocument для XML): mixed container (string type). И ещё один вспомогательный метод для ядра, который подскажет ядру какой же контент из всех накопленных в итоге выводить: void prefer (string type).

Например, генерация обычной HTML-страницы из накопленных XML-данных будет выглядеть так:

$dom = coren::container(‘maindom’);

$dom->documentElement->appendChild($dom->createElement(‘data’));

…

coren::prefer (‘maindom’);

А вывод картинки мог бы выглядеть так:

while (!eof($file))

{

$chunk = read($file, …);

coren::content(‘imager’, $chunk);

}

coren::prefer(‘imager’);

Эффекты здесь такие:

* DOM-документы для XML создаются тогда и только тогда, когда им действительно кто-то пытается сплавить данные. Что экономит ресурсы когда XML и не нужен вовсе (например, генерация картинок или AJAX-JSON-данных).
* Значительно упрощается код ядра, так как из него удаляется всё про DOM, XML, XSLT, и переносится в модули.
* Можно создавать любые типы результирующих данных без модификации кода ядра, а также заменять имеющиеся типы новыми (в т.ч. сторонними) обработчиками.
* Генератор рассылок в этом случае может получать оформленные письма методом запроса к тому же самому сайту, на котором он и работает, получением ответа (в т.ч. multipart) и помещением его в очередь рассылки.

Благодаря модульной системе мы избавимся от нынешнего костыля, при котором для вывода альтернативного контента (не-XML/HTML) приходилось «затыкать» ядро и выводить всё самому; при этом ресурсы на генерацию и обработку XML полностью или частично всё равно тратились.

А вообще, всё то же самое можно сделать без ввода новых методов ядра, а просто с помощью событий: событие-запрос для получения контейнера, событие-оповещение о появлении нового контента.

А вот указание ядру какой же вывод из всех доступных сделать нужно реализовывать иначе. Для этого нужно в модулях обработки контента ввести конфиг «приоритет вывода». Если к ним поступает (значимый?) контент, то они не только создают необходимые для себя контейнеры, но и регистрируются в ядре с указанием своего приоритета. В конце работы ядро передаёт управление тому модулю обработки контента, который зарегистрировался с бОльшим приоритетом. Остальные модули, даже если они и зарегистрировались (т.е. в них есть контент), игнорируются. Так, например, можно выставить бОльший приоритет для JSON-ответов, нежели для простого XML/HTML.

Модуль multipart-ответов может работать как основной, просто регистрируясь в ядре без всяких условий и с очень большим приоритетом. Он собирает контент с других модулей с помощью события «вернуть свои данные для multipart-контента» (т.е. все модули должны уметь упаковать себя в, например, MIME-часть), упаковывает в общий multipart, и подаёт на вывод.

Даже модуль сборки XML ответа и обработки его с помощью XSLT-шаблонов может иметь настраиваемую функциональность впихивания в ответ определённых данных, таких как набор полученных привилегий (берётся от ядра), идентификационные данные посетителя (берётся событием-вопросом) либо пометка об анонимном входе, и т.п.

## Предположительный API ядра

Ядро оформлено как класс с именем “coren” (от Core Numeri). Класс статический, все методы в нём статические. Создание экземпляров не предполагается. Класс в данном случае лишь выполняет роль namespace'а.

Методы общего назначения:

bool load\_class (string classname)

string current\_stage ()

string current\_module ()

string current\_method ()

string version ()

Методы управления событиями:

void handler (string method, string event, array map)

mixed event (string event, mixed data)

Методы привилегий:

void grant (array privileges)

bool have (string privilege)

## Слоты и имена данных (для XML)

## Главный конфиг ядра и сайта

## Обработка ошибок и исключений

## Система привилегий и прав доступа

## Динамическая загрузка классов и файлов

## Создание модулей «на лету»

## Статистика выполнения

## Утилиты (утилитные статические классы)

## Хранимые процедуры в БД

# Нерешённые задачи

## Определение типа запроса

Как определять какой тип запроса к нам поступил, и в каком формате мы должны подать вывод (то есть какому модулю скармливать данные)? А также какие модули вообще запускать?

Также: как определить какие алгоритмы вообще выполнять на основе выбранного типа запроса и его вывода? То есть, по сути, не станется ли так, что для разного типа вывода потребуются свои модули не только вывода, но и выполнения?

### Решение №1

Все данные накапливать в DOM-контейнере в ядре. Затем модули обработки вывода сами определят каким способом это отформатировать.

Недостаток: потребность создавать DOM-контейнер даже для бинарных данных (например, картинок), хотя в них этот контейнер в итоге игнорируется; также DOM неактуален для простых текстовых сгенерированных файлов (CSS, JS и т.п.).

### Решение №2

## Двухсторонняя связь компонентов

Пока неясно как это сделать. Очевидно, что клиентская часть компонента должна иметь взаимодействие (отправлять и получать данные) непосредственно серверному своему напарнику, а не ядру в целом.

# Прочие заметки

## Назначение\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_