УДК 004.7

ОБЩИЕ РЕКОМЕНДАЦИИ ПО ПОВЫШЕНИЮ ЭФФЕКТИВНОСТИ ПРИ РАЗРАБОТКЕ WEB-САЙТОВ И ПРИЛОЖЕНИЙ

Гутовский Дмитрий Игоревич[1], Добрынин Владимир Николаевич[2]

[1]Аспирант;
ГБОУ ВО МО «Университет «Дубна»,
Институт системного анализа и управления;
141980, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: mfhnb@mail.ru.

[2]Профессор;
ГБОУ ВО МО «Университет «Дубна»,
Институт системного анализа и управления;
141980, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: arbatsolo@yandex.ru.

В данной статье проводится анализ различных технологий, используемых в WEB-разработке, и даются рекомендации и обоснования для эффективного выбора нужных.

Ключевые слова: CMS, сайт, WEB, HTML, CSS, PHP, управление содержимым, вёрстка, Интернет, безопасность, сокрытие информации, визуальный редактор.

GENERAL RECOMMENDATIONS FOR IMPROVING EFFICIENCY IN THE DEVELOPMENT OF WEB SITES AND APPLICATIONS

Gutovskiy Dmitriy Igorevich[1], Dobrinin Vladimir Nikolaevich[2]

[1]Graduate student;
Dubna State University,
Institute of the system analysis and management;
141980, Dubna, Moscow reg., Universitetskaya str., 19;
e-mail: mfhnb@mail.ru.

[2]Professor;
Dubna State University,
Institute of the system analysis and management;
141980, Dubna, Moscow reg., Universitetskaya str., 19;
e-mail: arbatsolo@yandex.ru.

This article analyzes the various technologies used in WEB-development and provides recommendations and justification for the effective selection of the desired.

Keywords: CMS, site, WEB, HTML, CSS, PHP, content management, typesetting, the Internet, security, information obfuscate, editor with GUI.

Введение

Данная работа является продолжением статьи «Общие рекомендации для проектирования и реализации *WEB*-сайтов». Наиболее подробно рассмотрены задачи и проблемы, возникающие при реализации *WEB*-сайтов различной направленности и сложности. Актуальность *WEB*-технологий, на сегодняшний день неоспорима, главным образом, благодаря кроссплатформенности на стороне конечного пользователя. *WEB*-технологии, с каждым днём, охватывают, всё-более обширный круг задач, однако, выбор этих самых технологий, весьма затруднителен, так как их число обширно, а стандартов, зачастую нет (особенно, для серверной части).

В статье дан набор общих рекомендаций, позволяющих, наиболее обоснованно выбрать и применить ту или иную группу средств и технологий разработки. Все конкретные ситуации, а также точные названия различных готовых решений, даны лишь для того, чтобы обоснования и объяснения не выглядели абстрактными и голословными. Рассмотренные решения применимы на практике в различных ситуациях, и не являются призывами к конкретным действиям.

Цель

Облегчение выбора тех или иных технологий *WEB*-разработки для более эффективной реализации конечного продукта.

MODx-концепция (кратко)

Все *CMS*, вне зависимости от их направленности, можно поделить на две основных группы: одни придерживаются концепции *MODx*, а другие – нет. *MODx* взят как пример, а не как единственный представитель данной концепции, хотя, *MODx*-подобных *CMS* меньшинство. Основным отличием *MODx*-подобных *CMS*, от их стандартных аналогов, является то, что *API MODx*-подобных *CMS* придерживается принципа разделения ответственности (*Single Responsibility Principle*). В *MODx* нет функций, которые отвечают за вывод целой секции контента, например, как виджет боковой панели в *WordPress*. Дело в том, что в стандартной *CMS*, под разные группы содержимого, есть конкретные функции вывода, а в *MODx* есть свой язык-интерпретатор для шаблона. Таким образом, для вывода различных секций шаблона, необходимо задать переменные и типы данных, которые будут выводиться через них, аналогично императивным языкам программирования. Поведение этих секций, определяется разработчиком шаблона сайта, а не предопределёнными функциями *CMS*.

Например, для вывода названия сайта в *CMS WordPress*, нужно использовать функцию *bloginfo*, с параметром name. В *CMS MODx* тоже есть зарезервированные поля и настройки, однако, можно создать свои, и базовые вообще не использовать, так как пользовательские дополнительные поля (переменные шаблона), имеют те же права, что и основные (за редким исключением). В *MODx* не нужно запоминать множество функций и правил работы с ними, так как сам графический интерфейс данной *CMS*, позволяет не только управлять содержимым сайта, но и создавать функционал, и для создания тех же переменных шаблона, не нужно учить конкретных функций, всё задаётся через опции самой *CMS*. Например, требуется вывести миниатюру страницы, не нужно искать конкретных функций, которые выведут это, или позволят создать новый тип данных, если такого нет в *CMS*. Нужно просто задать необходимые параметры, в соответствующем разделе панели управления. Таким образом, в *MODx* есть полноценная среда разработки. Вне зависимости от поведения той или иной секции шаблона, всё реализуется через одни и те же сущности (поля, чанки, сниппеты и т.д.), различаются только настраиваемые параметры и выбранные типы. Это не только облегчает разработку шаблона, но и улучшает совместимость между разными версиями таких *CMS*. Однако, в *MODx*-концепции есть явный недостаток, заключающийся в отсутствии возможности автоматической установки готовых шаблонов. Это обусловлено отсутствием чётких правил и привязок. Такая абсолютная «гибкость» *CMS*, помогает при разработке сложного сайта «под-ключ», но не даёт возможности построить свой сайт необученным людям, без использования программирования, «одной мышкой». Таким образом, гораздо легче реализуются, даже самые экзотические требования заказчиков, но только в конечном продукте, а не в «полуфабрикатах», для самостоятельной сборки сайта произвольными людьми.

Допустим, в той же CMS *WordPress*, которая не придерживается *MODx*-концепции, есть определённое расположение тем и плагинов по умолчанию, относительно корня сайта, например, для шаблонов это /wp-content/themes/название темы. В *MODx* такого нет. Разработчик сам решает, где и что хранить, это добавляет гибкости, но даже на этом этапе понятно, что нет единых стандартов, относительно конечного шаблона, есть только стандарт для разработки, как и в языке программирования.

Допустим, с той же миниатюрой страницы, нужно просто зайти в «элементы», «дополнительные поля», и создать новую переменную шаблона, указав её название и выбрав тип данных, например, изображение. В зависимости от этого типа будут разные интерфейсы для конечного пользователя, например, для изображения, будет выдаваться диалоговое окно, позволяющее выбрать файл, а для

типа «текст», будет выдаваться строка для его ввода. Допустим, переменная для миниатюры названа thumbnail. В этом случае нужно будет просто вставить в соответствующую секцию вёрстки, вызов этой переменной [[*thumbnail]] (*MODx Revolution*). Если нужно задать какие-нибудь текстовые данные, например, название этого изображение, то можно создать ещё одно поле, вызов которого и способ создания аналогичны предыдущей ситуации. Изменится только тип переменой, который выбирается в самом графическом интерфейсе *CMS*, при разработке шаблона. Понятно, что и поведение переменных можно поменять, например, будут ли они одинаково выводиться на страницах, или при определённых условиях, или разные для каждой страницы, это не важно, так как базовые сущности, типа переменных, остаются без изменений. Все функции *MODx*-подобных *CMS* направлены не на вывод конкретного содержимого, а на интерпретацию шаблонов и модулей, с теми параметрами, которые указал разработчик.

Более подробную информацию о *MODx*-концепции можно получить в статье «Определение основных концепций *CMS*» и в предыдущей статье «Общие рекомендации для проектирования и реализации *WEB*-сайтов».

Задача сокрытия информации о серверной части WEB-сайтов и приложений

Для защиты сайтов рекомендуется скрывать (или подменять) информацию о серверной части. Клиент не должен знать о том, при помощи каких программ и технологий сгенерирована итоговая страница, которую он видит в браузере. В противном случае это может привести к тому, что злоумышленник, узнав про вышеупомянутые технологии, воспользуется их уязвимостями, облегчив себе задачу неправомерного проникновения в систему.

Помимо стандартных рекомендаций, описанных в предыдущий статье, таких как скрытие модели сервера, языка приложения, *CMS* и т. д., необходимо учитывать те факторы, которые могут косвенно разоблачить компоненты серверной части.

Одним из наиболее слабых мест, через которые ведутся атаки, являются уязвимости *CMS*. Полный отказ от распространённых *CMS* невозможен, так как не для каждого проекта имеется бюджет и сроки, позволяющие нанять специалистов, которые напишут всё «с нуля». Безусловно, что даже идеальное сокрытие информации о *CMS* не обеспечит 100% безопасности, однако сильно снизит риски.

Одним из факторов, раскрывающих информацию о *CMS*, является специфическое именование стилевых селекторов (чаще – классов). Например, имеется абстрактная система, под-названием *My-CMS*. Страница, генерируемая этой системой, имеет блоки с классом *myclass*. Увидев данные названия классов, специфические для конкретной *CMS*, возникает предположения о её модели, а иногда и версии. Если, во время написания шаблона, можно попытаться переопределить классы, которые будут генерироваться самой *CMS*, на отличные от стандартных (хотя в некоторых *CMS*, это весьма сложно сделать), то для работы визуальных редакторов возникает ряд других проблем. Рассмотрим подробнее.

В абстрактной *CMS My-CMS* имеется визуальный редактор для работы с содержимым. Такой редактор может работать по двум основным методикам: первая методика заключается в том, что в шаблоне содержится файл стилей, необходимый ни только для отображения самого шаблона, но и для реализации стилевых эффектов редактора. Допустим для того, чтобы выровнять текст по левому краю, визуальный редактор, идущий в составе *My-CMS*, прибавляет элементу с этим текстом класс *classleft*. Реализация (стилевые правила) класса *classleft* содержатся в *CSS*-файле, который идёт с шаблоном для этой *CMS*. Этот файл подключается вместе с основными стилями шаблона и таким образом, происходит активация эффекта, который был выбран в редакторе. Вторая методика заключается в том, что стилевые правила, выбранные в редакторе, записываются непосредственно в тэге с конкретным элементом, через атрибут *style* (так-называемые *In-Line* стили). Это не является нежелательной конструкцией, так как стилевые правила записываются на языке *CSS*, а не *HTML*-атрибутами и тэгами. Подробнее, про нежелательные стилевые конструкции можно посмотреть в предыдущей статье, в разделе «Задача унификации и оптимизации вёрстки».

Основными недостатками первой технологии работы визуальных редакторов являются:

- Необходимость включения в шаблон *CSS*-файла со стандартными стилевыми эффектами для редактора конкретной *CMS*. Конечно, можно прикрепить стилевой файл в комплект к самому редактору, однако, данный способ применим не для всех эффектов. Например, в редакторе имеется кнопка для выделения цитат. Эффект от такого выделения, в дизайнах различных шаблонов, может быть отображён по-разному. Также, такой файл стилей может иметь в себе правила, пересекающиеся с правилами в самом шаблоне, а также со сторонними стилевыми библиотеками и плагинами.
- Вытекающий из предыдущего пункта недостаток, заключающийся в возможном конфликте стилей.
- Несовместимость версий. Редакторы, идущие с разными версиями одной и той же *CMS*, могут иметь различные названия классов, для реализации тех или иных стилевых эффектов.
- Файл со специфическими стилями редактора, рассекречивает информацию о *CMS*.

Последний недостаток является одной из основных проблем, приводящих к рассекречиванию *CMS*, а следовательно – открытию уязвимостей для злоумышленников.

Одним из наиболее простых решений, позволяющих устранить вышеупомянутые недостатки, является использование стороннего редактора, работающего при помощи *In-Line* стилей, либо – создание плагина-перехватчика, который отключает *CSS*-файл для эффектов редактора, а после этого, подменяет классы, приписанные этим редактором на правила, реализующие эти классы, записывая эти правила в конкретных тэгах, через атрибут *style*. Писать In-Line стили для фиксированных (заранее известных) частей шаблона, например, меню, шапки и т. д. – не целесообразно, так как ведёт к запутыванию кода. Если для генерации фиксированных элементов (например, пунктов определённого меню), встречаются специфические классы конкретной *CMS*, их нужно постараться переопределить. *In-Line* стили целесообразны только для содержимого, стили которого задаёт пользователь (как-правило – то, что и редактируется визуальным редактором).

Основным недостатком второго (*In-Line*) способа работы редакторов является то, что затрудняется возможность быстрого изменения стилевых правил для группы элементов в контенте. Однако, здесь есть нюанс, который заключается в том, что стандартные классы редакторов (методика 1), отвечают за конкретную функцию. Например, нужно сделать жирный текст, для этого есть класс с конкретным правилом, если нужно сделать текст ещё и курсивным, то для этого есть другой класс. Это обусловлено тем, что пользователь может применять эффекты в любых комбинациях и они не связаны между собой. Если пользователю требуется объединить несколько эффектов в одну группу, то некоторые редакторы реализуют это так, что появляется возможность создать собственный класс, и прибавлять его к нужным элементам. Название этого класса – произвольно, а следовательно – не раскрывает модель *CMS*. Также, такое решение проблемы с быстрой подменой стилей, не как не противоречит *In-Line* методике работы редактора, ведь не кто не запрещает приписать тэгу атрибут *style*, содержащий нужные правила, а также добавить атрибут *class*, содержащий, в качестве значения, названия собственных классов. Реализации этих классов могут приписываться к нужным страницам в тэге *style* или может генерироваться временный отдельный файл стилей, который будет находиться в кэше сайта (на стороне сервера), и подключаться к нужным страницам. Из этого следует, что можно использовать *In-Line* подход для одиночных фрагментов и функций, но не запрещать создавать собственные классы, которые содержат нужный набор эффектов, имеют произвольные названия, а также, приписываются только к нужным для пользователя элементам. Таким образом, решается проблема устранения основных недостатков у этих двух методик работы визуальных редакторов.

Нелишним будет создание файлов-заглушек в основных директориях *CMS*. Допустим, у *CMS WordPress* основной конфигурационный файл имеет штатное название *wp-config.php* и содержится в корневой папке самой *CMS*. Большинство серверов настроено так, что при вызове файлов серверной части, через стандартный *HTTP* (или *HTTPS*) протокол, отобразится только результат выполнения их кода. Так как конфигурационные файлы, сами по себе, обычно не выводят информацию на экран и не генерируют никакой *HTML*-код, то браузер получит пустую страницу. Однако, такая пустая страница скажет злоумышленнику о том, что тот или иной конфигурационный файл имеется на сайте, а значит можно предположить использование определённой *CMS*. Чтобы запутать злоумышленника, нужно создать имитации конфигурационных файлов, характерных для различных *CMS*, и расположить их в соответствующих папках. Данные имитационные файлы могут содержать какой-нибудь незначащий код, или быть пустыми. Такие файлы никак не отразятся на производительности *WEB*-приложения,

так как ни к чему не привязаны и могут быть вызваны, только намеренным обращением пользователя. Однако, их наличие, запутает ни только пользователя, но и сервисы определения *CMS*, что положительно скажется на безопасности системы. Так как, при правильной настройке сервера, сам код серверной части не должен быть передан на сторону клиента, то пользователь (и сервисы определения бэкэнда), могут только предполагать ту или иную серверную часть. Можно, также создать файлы-заглушки, характерные для *CMS*, написанных на языках, отличных от того, на котором работает конкретный сайт. Однако, сервер не обязан распознавать произвольные типы файлов, а значит, может выдать их код на клиентскую часть. Настройка сервера может быть недоступна для разработчика *WEB*-приложения, так как может быть куплен настроенный хостинг, без прав на сервисные операции над конкретной площадкой. Если при обращении к конфигурационным файлам конкретной *CMS*, выдаются какие-то специфические ответы, то было бы нелишним вписать эти ответы в файлы-заглушки, имитируя тем самым, ответ приложения, написанного на другом языке.

Остальные компоненты, которые необходимо скрывать, зависят от многих факторов. Здесь невозможно дать единых рекомендаций, так как у каждого программного продукта есть свои конкретные особенности. Однако, в большинстве *CMS* есть штатные средства, позволяющие изменить названия системных папок, пути к ним, а также есть другие средства маскировки. По возможности лучше постараться, не только скрыть информацию о конкретной *CMS* (и других компонентах серверной части), но и заменить её на ложную. Например, использовать *In-Line* подход в контенте, а для фиксированных модулей (например, для меню сайта), переопределить классы на те, которые характерны для другой *CMS*. Однако, далеко не во всех *CMS* имеются штатные средства для переопределения модулей, и сделать это без переписывания самой *CMS* – весьма сложно. Изменять *CMS* – не рекомендуется, так как эти изменения будут утеряны при обновлениях и могут не поддерживаться штатными плагинами и дополнениями.

Задача обеспечения заменимости компонентов в шаблоне

Помимо основной части содержимого страницы, существуют компоненты, которые находятся на фиксированных позициях (заранее определённых в шаблоне), но содержимое этих компонентов может меняться, в зависимости от выбора пользователя. Например, в шаблоне предусмотрена позиция под логотип сайта. Несмотря на то, что позиция нахождения этого логотипа предопределена, сам логотип выбирается пользователем. Наиболее простая ситуация, когда логотип состоит из одного элемента, например, изображения. В этом случаи, можно определить максимальные параметры длины и высоты этого элемента, не давая объекту с любым разрешением деформировать блок и окружающую вёрстку. Однако, бывают ситуации, когда в подобных блоках могут содержаться несколько частей, например, блок с названием компании, а также блок с девизом. Похожая ситуация с составным содержимым, внутри одного блока, может встретиться в разных участках шаблона, ситуация с логотипом взята для примера. Плюс к этому, точные линейные размеры, для того же названия не известны, всё зависит от количества букв. Хотя, максимальное ограничение может быть введено. Рассмотрим ситуацию, когда название компании находится на определённом месте, а девиз находится справа от названия, с определённым отступом. Очень часто, подобные блоки верстаются с применением абсолютного позиционирования, однако, не кто не может гарантировать, что данный шаблон будет использоваться только в одном проекте, и что эти блоки будут статичны. Абсолютное позиционирование, практически лишает пользователя возможности изменения информации в этих блоках, даже если предусмотреть эту функциональность на уровне серверной части. Наиболее правильным решением данной проблемы будет использование свойств таблиц для данных блоков.

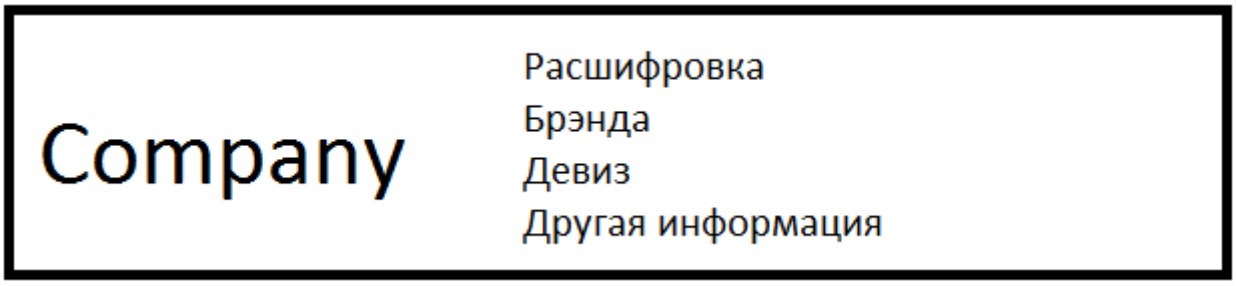

Рис. 1. Схематичное изображение составного блока с логотипом

Выше приведён абстрактный, схематичный пример такого блока. Изменение длины названия компании, деформирует блок с логотипом, в случае применения абсолютного позиционирования. Конечно, это – далеко не единственная ситуация, в которой возникают подобные проблемы, однако, абсолютное позиционирование, зачастую неоправданно применяется, что может облегчить работу верстальщика, но усложнить работу серверного программиста, которому придётся предусмотреть ситуации динамического изменения содержимого, исправляя вёрстку.

Ещё один пример, который демонстрирует плохую заменимость компонентов, заключается в нетипичном использовании тех или иных свойств. Это может быть оправдано, примеры можно посмотреть в предыдущей статье, в разделе «Задача унификации и оптимизации вёрстки». Однако, встречается неоправданное использование тэгов и стилей не по прямому назначению. Допустим, имеется некий блок, содержащий в себе текст и изображение. При этом текст не как не пересекается с изображением, а находится, например, сверху. Ниже приведён схематичный пример.

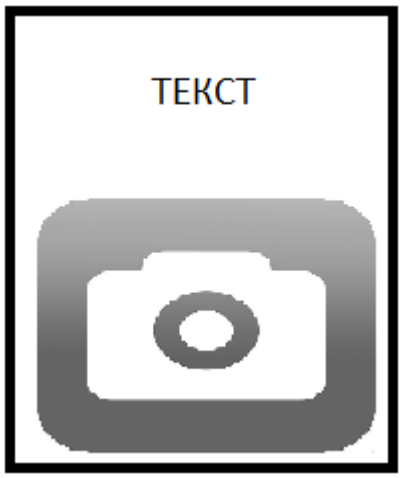

Рис. 2. Схематичный пример составного блока текста с изображением

В данной ситуации наиболее логичным было бы поместить изображение в тэг *img*, однако, зачастую изображение делается фоном блока с текстом (через свойство *background* в *css*), после чего, появляются проблемы, заключающиеся в управлении размерами данного изображения. Также, приходится писать атрибут *style* прямо в тэге для того, чтобы менять это «фоновое» изображение динамически, хотя этот блок является определённым заранее в шаблоне, а не частью произвольного контента, следовательно, там нет смысла писать *style* в конкретном тэге. При использовании тэга *img*, не возникло бы проблем ни с атрибутом *style*, не с подгоном размеров изображения. В данной, достаточно стандартной ситуации, такой ход с нетипичным использованием тэгов и стилей, не является оправданным, кроме случаев обрезки изображения при превышении линейных размеров, если это задумано в дизайне.

Также, необходимо выполнять «перестраховку» на стороне *CSS*. Возьмём пример с тем же логотипом. Если в том разделе, в котором находится этот логотип, на уровне *CMS*, предусматривается ограничения по количеству вводимых символов, то по крайней мере, пользователь не сможет деформировать окружающую вёрстку, вылетев за приделы конкретного блока. Однако, при дальнейшей модернизации сайта, может возникнуть необходимость сделать этот логотип другим разделом, на уровне *CMS*, при том, что в вёрстке, всё останется «как было», а ограничений в новом разделе не будет. Если не предусмотреть это в *CSS*, то вёрстка может быть деформирована «нештатным» контентом. Это свидетельствует о плохой заменимости компонента шаблона. Аналогично можно сказать, например, про меню на сайте. Если количество пунктов меню ограничено по задумке дизайнера, то это ограничение нужно сделать в *CSS*, чтобы не завязываться на опции конкретной *CMS*.

Большинство проблем заменимости, опять же, актуальны, по большей части, для *CMS* с не *MODx*-концепцией, так как в противном случае, нет дифференциации разделов, и ограничения индивидуально вводит разработчик. Хотя, перестраховку на стороне клиента лучше сделать, даже при использовании *MODx* и ей подобных *CMS*. Так как никто не может отвечать за все плагины, дополнения и библиотеки.

Безусловно, рассмотрены далеко не все примеры, ухудшающие заменимость компонентов, однако, данные ситуации встречаются часто, и наиболее наглядно демонстрируют тот факт, что во время посадки вёрстки на CMS, не изменять каркас вообще, получается не всегда. Хотя, в идеальном случае, каркас должен оставаться «как есть».

Задача рационального применения методологий типа «БЭМ»

Методология «БЭМ», название которой расшифровывается как Блок Элемент Модификатор, была разработана российской компанией Яндекс. Главной целью данной методологии является активное повторное использование кода. Для реализации этой цели, все сущности разделяются на блоки (отдельные, независимые друг от друга фрагменты вёрстки, например, блок меню), элементы (сущности, существования которых вне блока, не имеет смысла), а также модификаторы (классы с набором стилевых правил, позволяющие видоизменять блоки и элементы, в зависимости от ситуаций).

Рассмотрим пример. Имеется меню сайта. Само это меню будет блоком. Элементы этого меню, как и следует из названия, будут элементами. Сам элемент меню, находящийся вне блока меню, не имеет смысла. При этом сам блок с меню может быть использован несколько раз на странице, например, в «шапке» и в «подвале». В зависимости от того, в каком месте страницы находится это меню, его элементы могут иметь различный вид. Вот здесь нам и понадобится класс-модификатор. Заведя необходимые классы-модификаторы для разных ситуаций, требуется приписать нужные модификаторы к модифицируемым элементам. Таким образом, происходит отказ от каскадных селекторов в *CSS*. В традиционном случае, можно было бы написать, что есть блок с классом menu, в нём есть список *ul*, в нём есть элемент списка *li*, а в нём есть ссылка *a*. Таким образом идёт отсылка на каскад, однако в методологии БЭМ не рекомендуется использовать каскадные селекторы, чтобы блоки не зависели друг от друга и не было проблем с приоритетами и наследованием стилевых правил. Таким образом, и у блока меню, и у его элементов, имеются собственные классы. Также, в методологии БЭМ не рекомендуется использовать, в качестве селекторов, *ID* и тэги. Следует использовать только классы. Это следует автоматически, так как селектор по *ID* лишает возможности использовать фрагмент несколько раз, а селектор на тэг действует не избирательно. Однако, никто не запрещает использовать *ID* для других целей. Даже если элементу присвоены классы, и стили задаются только через них, можно присвоить элементам *ID*, например, для упрощения поиска этих элементов на странице, создания ссылок-якорей на конкретные участки, и т. д.

Благодаря тому, что в рамках данной методологии, стили не задаются для тэгов, можно будет не привязываться к конкретному *DOM*-элементу и подменять их, если этого требует семантика. Однако, тогда следует предусмотреть избыточное количество правил, или произвести наиболее полный сброс стилей, в самом начале *CSS*, так как стилевые свойства разных элементов, задаваемые браузером по умолчанию, могут отличаться. Плюс к этому, есть некоторый список *HTML*-тэгов, которые не поддаются большинству привычных *CSS*-приёмов. Например, тэг *select* и его варианты *option*, не стилизуются как большинство других элементов, таких как *span*, *div* и т. д. Это не единственный тэг с непривычной стилизацией, однако, таких тэгов уже не так много, но эта проблема остаётся актуальной на сегодняшний день, и она не решена в *HTML*5 и *CSS*3. Поэтому, иногда приходится нарушать семантику, подменяя такие элементы, или скрывать их, добавляя на страницу дополнительные тэги. Это делается для достижения нужных стилей. Даже плагины, которые позволяют стилизовать подобные тэги, в большинстве случаев, работают путём добавления дополнительных элементов, просто их не пишут в искомую вёрстку, а они добавляются плагином автоматически, при загрузке страницы, и *DOM*-структура строится вместе с этими добавленными элементами.

Не стоит забывать о том, что большое количество слишком сложных (по названию) классов на странице, может ухудшить *SEO*-оптимизацию. Также, в браузерах имеются ограничения на длину названия классов, вложенность каскадов и количество классов, присваиваемых конкретному элементу.

Плюсом данного подхода является независимость и возможность повторного использования блоков (для чего и создавалась эта методология). Однако, её применение оправдано, далеко не везде. Рассмотрим пример. Требуется изготовить шаблон для абстрактной *My-CMS*. Вёрстка для этого шаблона была сделана по всем канонам БЭМ. Остановимся на том же меню. У *My-CMS* есть функция, отвечающая за вывод элементов меню. При этом она генерирует список *ul* и элементы *li* (а также

ссылки в них), самостоятельно. Естественно, что класс, генерируемый для этих *ul* и *li*, будет отличаться от того, который написан в вёрстке (если не предусмотреть заранее). Также, класс может вообще отсутствовать. Если бы был каскадный селектор, то проблемы бы не возникло. Был бы блок с классом *menu*, а элементам, лежащим внутри (непосредственно самим тэгам), задавались бы правила. Но так как у этих элементов, в исходной вёрстке были свои классы, то сгенерированное *My-CMS* содержимое, будет отличаться от исходного, своим видом. Конечно, эта проблема решается переопределением самих модулей, а точнее – их шаблонов. Однако, далеко не в каждой *CMS* имеется штатная возможность легко выполнить такие переопределения, а про множество плагинов, с их особенностями, вообще не имеет смысла говорить. Под каждое серверное приложение не подстроишься.

Даже в том случае, когда удалось произвести все переопределения, то полное соблюдение методологии БЭМ, в ряде случаев – всё равно невозможно. Там, где пользователь вводит произвольный контент, вёрстка и стили генерируются динамически, как-правило, всё равно придётся ссылаться на каскад. Иначе, придётся писать собственный редактор, эта проблема, также пересекается с задачей сокрытия информации о серверной части.

Таким образом, можно сделать вывод о том, что методология БЭМ оправдывает себя в очень больших проектах, типа того же Яндекса. Серверная часть (в том числе и *CMS*), для таких проектов, как-правило, пишется индивидуально. Соответственно, проблемы состыковки не возникают. Также, в проектах такого масштаба, повторное использование блоков является, действительно необходимой задачей. В более универсальных и менее масштабных проектах, например, при создании тем для *CMS* общего назначения, создании плагинов для них и так далее, методология БЭМ будет сильно усложнять проект, из-за вышеуказанных проблем. Также, данную методологию не получится соблюсти полностью, если используются сторонние библиотеки, типа *bootstrap*, так как они (в том числе и сам bootstrap), не имеют отношения к БЭМ и не обязаны подчиняться этой методологии.

Правда, не следует забывать о том, что каскады обрабатываются медленнее одиночных селекторов, хотя, данная проблема уже не сильно актуальна, даже для мобильных браузеров. Во время использования каскадной модели основная проблема заключается в пересечении правил. Для того, чтобы этого избежать, нужно следовать принципу разумности, и делать каскад от того элемента, который реально является обособленной сущностью, а не завязываться, чуть-ли, не от *HTML*. Глубокие каскады – другая крайность, поэтому, делить вёрстку на независимые (по смыслу) блоки, всё равно нужно, вне зависимости от используемой методологии.

Однако, если возникает необходимость создать собственную библиотеку с готовыми блоками, то для этих целей, БЭМ действительно является удобным решением. Части такой библиотеки можно будет использовать позже, в проектах различной величины и направленности. В итоге получается, что данная методология, действительно удобна в некоторых ситуациях, но не нужно стремиться к её полному соблюдению. По аналогии с тем, что после появления блочной вёрстки, не следует отказываться от основных преимуществ табличной, и следует использовать её свойства из условий удобства, пускай и в неявном виде, а через *CSS*, при этом там, где таблицы – штатный способ вывода информации (например, в прайс-листе), то там следует писать таблицу явно, а не делать её из блоков.

Методология БЭМ используется не только для приведения стилей к определённому виду, но и для того, чтобы можно было не редактировать *HTML* напрямую, а описывать структуру страниц в удобном виде, используя собственную разметку (например, полученную на основе *XML*), а после этого, автоматически собирать страницу, при помощи, так-называемых БЭМ-парсеров. Правда, это также оправдано только в очень больших проектах, с частым переопределением блоков, а также, если вариантов различных блоков слишком много. Но при автоматической сборке, уже нельзя нарушать единую стилистику БЭМ, поэтому, при использовании сторонних плагинов, например, слайдеров, их придётся переписать, если они изначально не подчиняются БЭМ. Хотя, на БЭМ, всё равно нет единого стандарта, это методология, а не технология, поэтому, как называть блоки, каким знаком отделять названия элементов и т. д. Это не где не регламентируется, поэтому с автосборкой, всё равно придётся дорабатывать сторонний код, под конкретную реализацию БЭМ.

Задача семантически-верной вёрстки

При вёрстке *WEB*-страниц, может возникать ситуация, когда код верен синтаксически, но не верен по смыслу (семантически). Наиболее наглядный пример такой ситуации можно привести, ни только из *WEB*, но и из вёрстки обычных документов. Допустим, при изучении пакета *Microsoft Office*, начинающий пользователь использует функцию заголовка, для выделения фрагментов текста, или наоборот, оставляет текст с параметрами, как обычный, но эмитирует стиль заголовка, вместо применения заголовка. Аналогичная ситуация встречается, например, при выделении текста жирным, редакторы многих CMS, ошибочно используют для этих целей тэг <*strong*>.

Большинство тэгов *HTML*, с точки зрения базового функционала, являются схожими. Безусловно, что имеются тэги, которые содержат специфические атрибуты и не могут быть заменены, например, тэг <*a*> для ссылки, или <*input*> для полей ввода в форме. Однако, львиная доля тэгов имеет схожее поведение по умолчанию, аналогичные правила стилизации и не содержит специфических атрибутов. Например, семантические тэги HTML5 могут быть заменены обычными <*div*>. Хотя и сами <*div*> можно заменить на неправильные по смыслу тэги, например, на <*header*>. Если даже, стили по умолчанию не сходятся, например, у <*p*> и <*h1*>, то такой вопрос легко решается с помощью *CSS*. Другими словами, любой блок текста, будь то заголовок или абзац, можно было бы заменить обычным <*div*> и это было бы корректно, с точки зрения базового синтаксиса. Однако, *HTML*-тэгов гораздо больше, и они содержат разные метаданные. Если вернуться к ситуации с *Microsoft Office*, то при создании, например, автоматического оглавления страниц, *Microsoft Word* смотрит именно на то, чтобы фрагмент текста был заголовком, а внешние эффекты не имеют значения.

Единственным исключением, когда можно использовать неверные семантические конструкции, является неполная стилизация некоторых элементов. Если посмотреть в спецификацию *DOM*, то на сегодняшний день, около 5% элементов отличаются по допустимым *CSS*-правилам от остальных. Например, тэг <*select*>. Или поле <*input*>, с типом *file*. В этих случаях допускается подобная имитация, и то, лучше прибегнуть к помощи *JS*.

Дело в том, что отображение пользователю стандартным браузером – не единственная задача *HTML*-тэгов. Во-первых, браузеры могут быть специализированные, например, речевые (для незрячих). Допустим, такой браузер должен продиктовать все пункты меню на странице, однако, если эти пункты будут завёрнуты в обычный <*div*>, а не в специфический <*nav*>, то браузер не сможет отличить список ссылок этого меню, от любого другого списка. Аналогичная ситуация и с *SEO*-оптимизацией, так как для поисковых роботов важно дифференцировать информацию. Одна из причин отказа от табличной вёрстки, в явном виде, как раз и есть *SEO*.

Зачастую, в *CMS* с не *MODx*-концепцией, особенно в тех, которые имеют специфическую направленность, например, блоги или форумы, имеется множество избыточных функций, которые бывает невозможно отключить отдельно. Допустим, имеется функция авто-собираемого оглавления страницы. Для того, чтобы выборочно удалить из собранного меню какие-либо пункты, нужно куда-то вводить признаки, по которым и будет производиться удаление. Для этого потребуется создать произвольные поля, но в не *MODx*-подобных *CMS* это может быть затруднительно, для некоторых разделов. В этом случае напрашивается неверное решение, например, добавление кнопок в редактор, которые создают видимую CSS-имитацию заголовков, например, из обычного <*p*>. Так как тэг такой имитации не будет заголовком, то в авто-оглавление он не попадает, однако, как уже было сказано выше, метаданные будут не верны, и многое из того, что не видно пользователю, но важно для сторонних приложений, станет некорректным. Также, и перепись самого редактора в не *MODx*-концепции, может быть затруднительна. Хотя, перепись определённых редакторов может быть нецелесообразна и в случае с *MODx*, но в ней не возникнет трудностей с созданием дополнительных полей в любом разделе, или создании сниппета, для включение нужного шорт-кода.

Остальные проблемы, возникающие в семантики вёрстки при использовании не *MODx*-подобных *CMS*, аналогичны тем, что описаны в других разделах, и связаны, в основном, с затруднением нормального переопределения тех или иных участков шаблона.

Задача копирования клиентской части с произвольного сайта

Периодически возникает задача полного, или частичного копирования клиентской части сайта, при том, что нет полноценного доступа к серверной. Механизм решения данной задачи идентичен, независимо от серверной части сайта, или программ для копирования. Поскольку речь не идёт о несанкционированном доступе к серверу, то можно получить то, что доступно на стороне клиента, в том числе, скопировать полностью вручную. Все программы подобного рода, только автоматизируют процесс для того, чтобы не переходить по множеству вложенных ссылок, копируя различные варианты страниц, а также не копировать каждый дополнительный файл отдельно. В теории всё должно копироваться без ошибок, а скаченный таким образом шаблон, после минимальной очистки кода, должен быть готов к повторной адаптации к другим *CMS*. Однако, на практике, такого не бывает, почти никогда. Рассмотрим некоторые проблемы, возникающие при таком скачивании.

Во-первых, далеко не всегда копируются все дополнительные файлы, например, шрифты, стили, скрипты и т. д. Дело в том, что способ подключения таких файлов, далеко не всегда стандартизирован. Если речь идёт о подключении стилей или скриптов, непосредственно с *HTML*-страницы, то тут, как правило, проблем меньше всего. Другое дело, что в самих дополнительных файлах могут содержаться отсылки к другим ресурсам. Большинство современных программ-копиров научились просматривать и дополнительные файлы (например, *CSS*), и копировать подключаемые в них элементы, например, фоновые изображения. Однако, бывают ситуации, что путь к тому или иному файлу может быть получен, только в результате определённых действий, а не простого перехода по ссылке. Например, после передачи на сервер конкретных параметров, приходит ответ, из которого, с помощью *JS*-скрипта составляется путь до нужного источника файлов. Могут быть и другие параметры, которых нет по умолчанию, например, индивидуальные настройки в файлах *cookie*. Ещё, не всегда переписывается иерархия файловой структуры. Часто бывает так, что после копирования, ссылка на файл, например, *CSS*, которая написана нестандартно, а, например, с помощью директивы *@import*, из других *CSS*, остаётся нетронутой. Так или иначе, копируются не все файлы, а также рушится иерархия файлов в вёрстке.

Вторым, не менее важным фактором, является плохая читабельность скопированного кода. Связано это, прежде всего с тем, что сама серверная часть может подключать для своих нужд множество элементов, которые не нужны в получаемой вёрстке. Особенно, это характерно для *CMS* с не *MODx*-концепцией. Эта же особенность приводит и к снижению безопасности. Пишется множество незначимой, с точки зрения повторного использования вёрстки метаинформации, которая нужна, например, для *SEO*-оптимизации конкретной страницы. Также, при выводе различного контента, обычно не предусматривают разборчивого форматирования кода, ведь это приводит к увеличению передаваемых по сети данных, а также – к дополнительной нагрузке на сервер.

В-третьих, ни все элементы вёрстки могут быть доступны изначально. Допустим, у копируемого *HTML*-шаблона есть особый вид страницы поиска. Однако, невозможно предусмотреть всех различных ситуаций, для задаваемых при поиске параметров. Самый простой пример – постраничная навигация, которая станет доступна, при превышении определённого лимита результатов поиска.

Четвёртый фактор заключается в неподготовленности самой копируемой вёрстки к различным ситуациям. Дело в том, что многие «перестраховки», делаются средствами *CMS*, а не *CSS*, чтобы не перегружать браузер клиента. Это особенно актуально, если имеется отдельная мобильная версия сайта. Допустим, что имеется меню, которое корректно отображается, только если в нём не более, чем n элементов. Количество элементов ограничено в самой *CMS*, а в *CSS*, на этот счёт нет никаких правил (подробнее, в разделе «Задача обеспечения заменимости компонентов в шаблоне»).

Шестая причина неполного копирования заключается в том, что все параметры темы не могут быть включены одновременно. Допустим, что на сайте доступно несколько схем цветового оформления, одновременно включаемого для всех страниц. Естественно, что будут подключены *CSS*-файлы, только с актуальными правилами. Допустим, что изменить их может только пользователь, обладающий соответствующим набором прав, которого нет у копирующего.

Седьмая причина заключается в переизбытке значимых, но индивидуальных правил. Например, множество правил генерируется редактором содержимого, для реализации различных пользовательских стилевых эффектов. Естественно, что при повторном использовании, переделанная тема может быть написана под *CMS*, совершенно с другим редактором, а правила отображения самого шаблона и

стили для редакторов, могут быть перемешаны, что усложнит задачу их разделения и последующей очистки кода.

Восьмая причина касается, непосредственно, настроек сервера. Допустим, что для защиты от различных атак, типа *DoS*, лимитируется количество запросов, производимых с одного *IP*-адреса, в единицу времени. В этом случае придётся подгонять скорость копирования, и возможно, использовать дополнительные *proxy*-сервера.

Плюс к вышеизложенному, не стоит забывать о том, что компоненты клиентской части, и *HTML,* и *CSS,* и *JS* очень часто подвергают сжатию. Это делается для снижения объёма передаваемых по сети данных, но это же работает и для защиты сайта. Дело в том, что под сжатием подразумевается ни только отмена форматирования текста, но и искажение иерархии кода, а именно, той её части, которая не повредит функциональности. Допустим, некоторые правила *CSS*, идентичные для различных селекторов, но описанные в оригинальном коде по отдельности (по смыслу), могут быть собраны при сжатии в одну группу, и перечислены через запятую, что приведёт к усложнению расшифровки *CSS*, даже если будет восстановлено форматирование. С *JS,* всё ещё сложнее, так как там могут и названия переменных изменить, на короткие, но бессмысленные, и функции сократить, если результат их работы не меняется. Всё это приведёт к тому, что при относительно сложном JS, его восстановление будет настолько труда-затратным, что проще будет написать новый скрипт. Естественно, что и комментарии, в большинстве случаев, будут удалены, кроме редких исключений, например, некоторых прагм для *Internet Explorer*, оформленных в виде *HTML*-комментариев. Поскольку, такие манипуляции по сжатию, в подавляющем большинстве случаев, делаются автоматически, при помощи соответствующих средств, то это не доставляет труда разработчику, но сильно улучшает эффективность работы сайта, плюс – даёт его защиту, и поэтому, это часто применяется, что вполне логично.

Несмотря не то, что копирование клиентской части сильно усложнено для повторного использования, оно сильно облегчено для ослабления защиты сайта. Дело в том, что клиенту, зачастую передаются сведения, которые он не должен знать. Особенно, данное явление характерно для *CMS* с не *MODx*-концепцией. Такие *CMS*, из-за жёсткой фиксации файловой структуры, вычисляются достаточно легко. А также, вычисляется множество установленных на них компонентов. Допустим, есть некоторый плагин, выполняющий реализацию фотогалереи на сайте. Зачастую происходит так, что необходимый для этого *JS*-файл, подключается прямо из папки с плагином, которая, в свою очередь, находится по стандартному (для конкретной *CMS*) адресу. Всё это, также ведёт к рассекречиванию серверной части. Плюс к этому, предъявляются повышенные требования к настройке сервера, чтобы тот не отдал связанные файлы, которые не влияют на готовую страницу, но могут содержать конфиденциальную (для сервера) информацию.

Таким образом, автоматическое копирование клиентской части сайта, позволяет вычислить серверную часть, намного точнее, чем многие стандартные авто-определители.

Рекомендации, для частичного решения данных проблем – прежде всего, нужно разрушить стандартную иерархию файловой структуры сайта. Более подробно можно узнать в разделе сокрытия серверной части.

Ещё одно применение автоматического копирования клиентской части – отладка редактора содержимого. Дело в том, что многие ошибки, особенно при проверке *HTML*-кода связаны, не с ошибками верстальщика, а с некорректной работой редактора в *CMS*. Тот код, который генерируется подобными редакторами, может содержать запрещённые тэги, закрывать тэги в неверных местах и т.д. Такое копирование позволяет вывести наиболее репрезентативную выборку ошибок редактора и создать плагины, заменяющие работу тех или иных его элементов. Однако, данный способ, также наиболее актуален для *CMS* с *MODx*-концепцией, так как в противном случае, создание произвольных полей может быть сильно осложнено, да и при замене редактора могут возникать конфликты *CSS* и *JS*-библиотек.

Автоматическое копирование, также может затронуть кэш сайта, что позволит провести наиболее углублённую статистику и обнаружить больше проблем безопасности, так как многие *CMS* меняют расширения некоторых файлов кэша, а те, в свою очередь, могут содержать фрагменты серверной части, и при некорректной настройке сервера, такие элементы могут быть переданы на сторону клиента.

Подготовка собственных тестовых площадок для отладки разрабатываемых приложений под различными средами разработки

Несмотря на то, что тарифные планы современных хостинг-провайдеров, достаточно демократичны, это утверждение можно отнести в первую очередь к массовым хостингам, обеспечивающим наиболее популярные потребности пользователей. Однако, существуют ситуации, в которых предлагаемые провайдерами условия размещения сайтов на хостинг-площадке не отвечают потребностям заказчика. Например, когда системные настройки требуется «тонко подгонять» или «менять на лету», устанавливать и конфигурировать отличное от типового ПО, и так далее. В этом случае альтернативой, предлагаемой хостинг-провайдерами, являются выделенные сервера (физические или виртуальные). Тарифы на аренду таких серверов довольно высоки, что может оказаться неприемлемым для физических лиц или малых предприятий, а для крупных корпораций, аренда хостинга у сторонней компании может противоречить соображениям безопасности. В таких случаях может быть рассмотрен вопрос создания собственного хостинга. При этом следует учесть ряд рекомендаций, связанных с обеспечением безопасности таких решений.

Одной из основных рекомендаций, является разделение проектов по виртуальным площадкам, изолированным друг от друга. Наиболее часто встречается разделение на площадку с действующим проектом (*production*), и площадку разработки (*development*). На последней в период разработки могут быть отключены какие-то ограничения сетевой безопасности (например, дана возможность удаленного управления настройками сервера).

Не менее важным условием является использование нестандартных портов (там, где это возможно). Однако, иногда без этого и не обойтись. Например, при запуске нескольких *WEB*-серверов, в приделах одного пространства ОС.

Ещё один немаловажный аспект заключается в том, что не следует открывать лишние каналы «контакта сервера с внешним миром». Например, на сервере настроена поддержка *SSH*. Существует протокол *SFTP*, который является эмуляцией *FTP*, через канал *SSH*. Если, открытие канала *SSH* не противоречит конкретной политики безопасности, то для передачи файлов, следует использовать *SFTP*, который будет получен, вместе с *SSH*, а не устанавливать поддержку, например, *FTP* в конкретной ВМ.

Ещё одной, достаточно простой, но часто игнорируемой рекомендацией, является то, что нужно закрывать удалённый доступ к сервисным утилитам. Например, для администрирования БД *MySQL*, на PHP платформах, часто используется утилита *PHPMyAdmin*. Доступ к ней должен быть открыт только тогда, когда ведётся активная разработка проекта. В дальнейшем доступ должен быть запрещён (или ограничен). Например, если нет возможности закрыть доступ к вышеупомянутой утилите, то нужно предоставить доступ только с внутреннего IP, попасть на который, можно будет только через виртуальную сеть, с другой ВМ. Или, хотя бы, заменить ссылку для этой утилиты, на нестандартную. Аналогичная ситуация и тогда, когда возникает необходимость дать возможность удалённого управления реальным (глобальным) сервером, например, по *SSH*.

Ещё одной рекомендацией, является то, что на виртуальных площадках и реальном сервере (управляющем этими площадками), желательно иметь ОС разного семейства. Например, если ВМ, которую невозможно достаточно изолировать, имеет ОС, семейства *Windows NT*, то на реальной рабочей станции, лучше использовать *Linux* и наоборот… Хотя, управление инфраструктурой ВМ (например, их резервное копирование), тоже должно осуществляться через отдельную ВМ. Тогда, важность этого фактора для реального сервера снижается. Управление и изменение настроек через реальный сервер, должно осуществляться в крайнем случае, а для штатного управления виртуальными площадками, лучше настроить ВМ, которая полностью автономна и не имеет каналов удалённого управления. На реальном сервере должен находиться образ этой машины, полностью настроенный и зашифрованный, чтобы, в случае неблагоприятного развития событий, можно было быстро восстановить штатное управление основными площадками.

Также, следует отменить удалённый вход с правами администратора на реальный (управляющий) сервер и на ВМ, через которую открывается доступ к управлению этим сервером (если таковая не закрыта). На этих машинах также не следует использовать стандартные способы аутентификации, например, при помощи пароля. Следует использовать многофакторную авторизацию.

Ещё одним желательным шагом является отдельная проверка целостности файлов (например, через контрольные суммы), представляющих конфигурационные настройки. В том числе следует ограничить права на их чтения, создав специальную группу, пользователи которой имеют право это делать. Для тех файлов, изменения которых могут понадобиться, но эти ситуации нетипичны, необходимо предусмотреть обязательное оповещения администратора, например, через электронную почту.

Также, после окончательной настройки, следует удалить все лишние приложения на управляющем сервере, вплоть, до автоматического менеджера пакетов в Linux (зависит от вида дистрибутива).

Далее, следует предусмотреть «критическую массу изменений», для конкретных реальных проектов. Допустим, происходит резервное копирование файловой структуры и БД сайта. Это копирование настроено по расписанию, в автоматическом режиме. Естественно, что дисковое пространство не безгранично, поэтому, хранятся только несколько последних версий архива. Для наименьших потерь, в случае атаки, копирование производится с большой частотой. Однако, в этом случае, возникает проблема, что произойдёт многократное копирование испорченных данных, затерев нужные. Это копирование необходимо каким-то образом остановить, когда оператора (администратора) нет рядом. В этом случае, следует учесть две основных ситуации, первая – были изменены (или нелегитимно прочитаны) конфигурационные файлы, вторая – произведена большая часть типичных изменений, нехарактерных по времени. Например, изменено слишком много контента в соответствующих таблицах БД, за небольшой промежуток времени. В этих случаях необходимо незамедлительно оповестить администратора и производить следующую архивацию, только после соответствующей команды.

Ещё следует ввести ограничения по времени для различных групп пользователей. Вряд ли, вы (или ваши сотрудники), начнут администрировать сервер в нехарактерное для этого время (например, в 3 часа ночи).

И, наконец, следует подменять информацию о сервере, например, установлен сервер *Apache* и ОС семейства *Linux*. Как минимум, это не должно определяться различными сторонними сервисами, а лучше, если будет ложная информация, например, сервер *NGINX* и ОС семейства *Windows*.

Заключение

В рамках данной статьи было проведено концептуальное сравнение различных групп *CMS* (раздел «*MODx*-концепция»), а также, были рассмотрены следующие типы задач:

- Задача сокрытия информации о серверной части *WEB*-сайтов и приложений.
- Задача обеспечения заменимости компонентов в шаблоне.
- Задача рационального применения методологий типа «БЭМ».
- Задача семантически-верной вёрстки.
- Задача копирования клиентской части с произвольного сайта.
- Подготовка собственных тестовых площадок, для отладки разрабатываемых приложений, под различными средами разработки.

Список литературы

1. Ромашов В. CMS Drupal / Система управления содержимым сайта. — М.: Питер, 2016. — С. 533.
2. Колисниченко Д. Выбираем лучший бесплатный движок для сайта / CMS Joomla! и Drupal. — М.: БХВ-Петербург, 2012. — С. 288.
3. Официальный сайт CMS WordPress для разработчиков. — [Электронный ресурс]. URL: https://codex.wordpress.org/ (дата обращения: 21.06.2019).
4. Официальный русский сайт CMS MODx. — [Электронный ресурс]. URL: https://modx.ru/ (дата обращения: 17.04.2019).
5. Официальный русский сайт CMS WiX. — [Электронный ресурс]. URL: http://ru.wix.com/ (дата обращения: 15.03.2019).