|  |
| --- |
| МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М. МАШЕРОВА»  Факультет математики и информационных технологий  Кафедра информатики и информационных технологий  Допущен к защите  «\_\_» \_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.  зав. кафедрой И и ИТ   \_\_\_\_\_\_\_ Е.А. Витько |
| ДИПЛОМНАЯ РАБОТА  МЕТОДЫ МИНИФИКАЦИИ HTML-ВЕРСТКИ  Специальность 1-31 03 07-01 Прикладная информатика  (программное обеспечение компьютерных систем)  Бакштай Виталий Викторович,  студент 4 курса, группы 41  Научный руководитель:  Адаменко Наталья Дмитриевна,  доцент кафедры информатики и информационных технологий  Витебск, 2018 |

Содержание

[**Введение** 5](#_Toc515324847)

[1. Теоретическая часть 7](#_Toc515324848)

[1.1. Постановка задачи 7](#_Toc515324849)

[1.2. Парсинг 7](#_Toc515324850)

[1.3. Antlr – краткие сведения 8](#_Toc515324851)

[1.4. Обход AST 9](#_Toc515324852)

[1.5. Visitor и Listener 9](#_Toc515324853)

[1.6. Минификации CSS 10](#_Toc515324854)

[1.6.1. Комментарии, пробелы, переходы на новую строку 10](#_Toc515324855)

[1.6.2. Переопределение свойств 11](#_Toc515324856)

[1.6.3. Использование сокращенных конструкций при задании свойств. 12](#_Toc515324857)

[1.7. Минификация HTML 12](#_Toc515324858)

[1.7.1. Комментарии, пробелы, переходы на новую строку 12](#_Toc515324859)

[1.7.2. Внутрений код 13](#_Toc515324860)

[1.7.3. Объединений JS/CSS файлов 13](#_Toc515324861)

[1.8. Минификация JS 13](#_Toc515324862)

[1.8.1. Объединение и сжатие констант 13](#_Toc515324863)

[1.8.2. Укорачивание локальных переменных 14](#_Toc515324864)

[1.8.3. Объединение и удаление локальных переменных 14](#_Toc515324865)

[1.8.4. Разворачивание if-конструкций, удаление недостижимого кода 14](#_Toc515324866)

[1.9. Построение сокращенных имен 15](#_Toc515324867)

[1.10. Поиска имен идентификаторов и классов 16](#_Toc515324868)

[1.10.1. HTML 16](#_Toc515324869)

[1.10.2. CSS 16](#_Toc515324870)

[1.10.3. JavaScript 16](#_Toc515324871)

[**Список использованных источников** 19](#_Toc515324872)

Реферат

Дипломная работа 48 с., 16 рис., 6 источников, 1 прил.

МЕТОДЫ ОПТИМИЗАЦИИ HTML-ВЕРСТКИ.

Объект исследования – HTML-верстка.

Цель работы – разработка библиотеки для минификации HTML-верстки и web-приложения для использования библиотеки.

Предмет исследования – методы минификации HTML-верстки.

Методы исследования – анализ и моделирование предметной области, методы объектно-ориентированной разработки, изучение литературы и технической документации по JavaScript, CSS, HTML, .NET CORE, ASP.NET CORE, Google Closure Compiler, CSSO, ANTLR 4.

Результатом является библиотека для минификации HTML-верстки и web-приложение, которое дает возможность загрузить HTML-верстку на сервер, выбрать доступные минификации из списка, сохранить минифицированную верстку.

Область применения: управление исходящим трафиком сервера, продвижение сайта в сети Интернет.

**Обозначения и сокращения**

JS – язык программирования JavaScript;

CSS – (Cascading Style Sheets) язык описания внешнего вида документа;

HTML – (HyperText Markup Language) стандартизированный язык разметки документов;

HTML-верстка – совокупность файлов HTML, CSS, JS и другие файлы используемые в гипертекстовых страницах;

MVC – шаблон проектирования Model View Controller;

UI – пользовательский интерфейс;

**Введение**

В наши дни на просторах интернета практический каждый сайт написан с использованием таких языков, как HTML, CSS, JS. Но не каждый разработчик задумывается над оптимальным написанием кода. Да это и не логично на этапе разработке сайта, так как думать над оптимальным написанием кода не всегда верно, поскольку это может понизить читаемость кода, и замедлить его разработку. К тому же на момент размещения сайта на серверах читаемость кода уже не настолько критична. Однако при этом возникает проблема с неоправданно большим размером загружаемых файлов для отображения страниц в браузере, особенно это заметно при открытии страниц с мобильного телефона.

Решение данной проблемы имеет важное практическое значение, так как чем меньше будет размер загружаемых файлов, тем быстрее они будут отображаться для пользователей. Поисковые системы также присваивают сайтам с меньшей скоростью загрузки более высокий ранг в списке выдачи для поискового запроса при прочих равных параметрах. Кроме того, на некоторых серверах (хостингах) ограничен размер хранимых данных, что также является важной причиной.

Основные пути решения проблемы оптимизации размера HTML-верстки:

* удаление ненужных пробельных символов, сюда входят также табуляция и переходы на новые строки;
* удаление комментариев, которые на момент размещения сайта не несут никакого смысла;
* переименование всех имен идентификаторов и классов в HTML-верстке;
* оптимизация изображений;
* объединение JS/CSS-файлов, так как один большой файл загрузится быстрее, чем много маленьких с таким же суммарным размером;
* минификация JS-файлов;
* минификация CSS-файлов.

Цель работы заключается в создании комплекса программ для минификации HTML-верстки. Разработанный комплекс должен содержать: библиотеку для минификации HTML-верстки, WEB-приложен для получения от пользователя архива, содержащего верстку. Он обеспечивает обработку данного архива библиотекой для минификации и предоставляет пользователю ссылку на скачивание минифицированной версии верстки.

Для достижения данной цели возникают следующие задачи:

* лексический анализ файлов HTML, CSS, JS;
* построение AST (abstract syntax tree) для каждого типа файлов;
* обработка полученных AST;
* рассмотрение методов оптимизации каждого типа файлов;
* нахождение в верстке изображений и их оптимизация;
* использование библиотеки Google Closure Compile для минификации JS-файлов:
* использование библиотеки CSSO для минификации CSS-файлов;
* использование библиотеки Magick .NET для оптимизации изображений.

Разработанная библиотека для минификации HTML-верстки также может использоваться в пост-обработке сервером страниц. Это увеличит время обработки запроса, но значительно уменьшит объем отправляемых данных клиенту.

# Теоретическая часть

## Постановка задачи

Имеется HTML-верстка, необходимо реализовать библиотеку способную минифицировать данную верстку по выбранным опциям, WEB-приложение предоставляющий UI для удобного взаимоействия с данной бибилиотекой.

Основный функции WEB-приложения:

* получение от пользователя архив содержащий в себе HTML-верстку;
* предоставлять пользователю возможность выбора опций для минификаций;
* после отправки верстки на обработку просматривать статус выполнения минификации;
* информировать пользователя о событиях, произошедших при обработке;
* после завершения предоставлять возможность скачать обработанную верстку на диск.

Основный функции библиотеки для минификации:

* получение набора файлов для минификации;
* миницикация с выбором отдельных опций;
* информирование о событиях при обработке;
* предоставление минифицированных файлов.

## Парсинг

Парсинг — процесс преобразования исходного кода в структурированный вид. Типичный парсер представляет собой комбинацию лексера и парсера. Лексер группирует символы исходного кода в значащие последовательности, которые называются лексемами. После этого определяется тип лексемы (идентификатор, число, строка и т.п.). Токеном называется совокупность значения лексемы и ее типа. В примере на рисунке 1.2.1 токенами являются “*sp”, “=”, “100”*. Парсер же из потока токенов строит связную древовидную структуру, которая называется деревом разбора. В данном случае “*assign”* является одним из узлов дерева. Абстрактное синтаксическое дерево или AST — дерево разбора на более высоком уровне, из которого удалены не значимые токены, такие как скобки, запятые. Однако существуют парсеры, в которых шаг лексирования и разбора совмещены.

Для описания различных узлов AST используются правила. Объединение всех правил называют грамматикой языка. Существуют инструменты, генерирующие код под определенную платформу для разбора языков на основе грамматик. Они называются генераторами парсеров. Например, ANTLR, Bison, Coco/R [4].

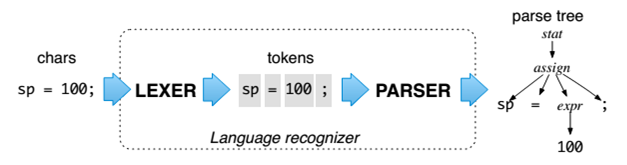


Рисунок 1.2.1 – Пример парсинга

## Antlr – краткие сведения

Для разбора файлов было решено воспользоваться фрэймворком Antlr (ANother Tool for Language Recognition – “ещё одно средство распознавания языков“). Одна из основных причин выбора, это наличие на сайте разработчика готовых файлов описания грамматики для большинства используемых языков. ANTLR позволяет создавать лексические и синтаксические анализаторы на различных языках (Java, C, C++, Python, C#, ActionScript, JavaScript, PHP) на основе грамматик [4].

Процесс создания анализатора можно описать следующим образом:

* создание грамматики для лексического анализатора
* создание грамматики для синтаксического анализатора
* отладка грамматики
* генерация классов лексического и синтаксического анализатора
* компиляция классов лексера(лексического анализатора) и парсера(синтаксического анализатора)
* использование классов для обработки текста

## Обход AST

Парсер преобразует исходный код в дерево разбора (дерево, в котором удалены все незначимые токены), называемом AST. Есть различные способы обработать такое дерево. Простой — заключается в обработке дерева при помощи рекурсивного обхода потомков в глубину. Данный способ применим только для совсем простых случаев, в котором существует немного типов узлов и логика обработки простая. В других случаях, необходимо выносить логику обработки каждого типа узла в отдельные методы. Это осуществляется с помощью двух типовых подходов (паттернов проектирования): Visitor и Listener [4].

## Visitor и Listener

В Visitor для посещения потомков узла необходимо вручную вызывать методы обхода дочерних узлов. При этом если родитель имеет три потомка, и вызвать методы только для двух узлов, то часть поддерева вообще не будет обработана. В Listener же методы посещения всех потомков вызываются автоматически. В Listener существует метод, вызываемый в начале посещения узла (enterNode) и метод, вызываемый после посещения узла (exitNode). Методы Visitor, в отличие от Listener, могут возвращать объекты и даже могут быть типизированными, т.е. при объявлении \*Visitor каждый метод Visit, будет возвращать объект AstNode, который в нашем случае является общим предком для всех остальных узлов унифицированного AST.

Таким образом, код преобразования дерева с помощью Visitor получается более функциональным и лаконичным за счет того, что в нем нет необходимости хранить информацию о посещенных узлах [4].

## Минификации CSS

### Комментарии, пробелы, переходы на новую строку

Каждый разработчик форматирует написанный код так, как ему удобно. В большинстве случаев описывается шаблон, как показано в Листинге 1.2.1

Листинг 1.2.1 Пример описание свойств селектора

div {

color: *black*;

padding: *10px*;

…

}

В большинстве случаев разработчик использует табуляции, переходы на новую строку, повторяющиеся пробелы. Все эти элементы «мертвый груз», который не требуется для корректного применения стилей к странице. Отсюда следует, что необходимо удалить их там, где это возможно сделать в исходном файле.

Каждый разработчик пишет комментарии при написании стилей. Но комментарии нужны только для того, чтобы повысить читаемость CSS-файла, а для отображения страницы они не играют никакой роли. Следовательно, необходимо удалить все комментарии в исходном файле.

Также часто разработчики ставят закрывающий символ свойства в последнем свойстве селектора.

Листинг 1.2.2 Пример с лишним закрывающим символом свойства

div {

color: *black*;

padding: *10px*;

}

Для парсера CSS браузера этот селектор будет обрабатываться также, как и в Листинге 1.2.3

Листинг 1.2.3 Пример без закрывающего символа свойства

div {

color: *black*;

padding: *10px*

}

Отсюда можно сделать вывод, что этот символ можно опустить в последнем свойстве селектора.

### **Переопределение свойств**

Встречаются так же ситуации, когда переопределяются уже написанные свойства для селекторов.

Листинг 1.3.1 Пример переопределения свойства селектора

div {

padding-top:*10px*

}

*/\* какой-то код \*/*

div {

padding-top:*20px*

}

Видно, что первую запись можно опустить. Отсюда возникает еще одна задача: убрать свойства селекторов, которые были переопределены.

Также, зачастую, некоторые свойства указываются по частям тогда, когда можно было использовать более краткую запись.

Листинг 1.3.2 - Пример задания свойства

div {

padding:*10px 20px 30px;*

padding-left:*40px;*

}

*/\*менее часто\*/*

div {

padding-top:*10px;*

padding-right:20px;

padding-bottom:*30px*;

padding-left:*40px*

}

Данные свойства селектора можно записать с помощью одного свойства (см Листинг 1.3.3)

Листинг 1.3.3 - Пример задания свойства

div {

padding:*10px* *20px* *30px* *40px*

}

Следовательно, нужно удалять переопределяемые свойства.

### Использование сокращенных конструкций при задании свойств.

Цвет разработчики задают так, как им удобно. Это словами (red, green, black и др.), функциями rgb ( rgb(255,0,0), rgb(255,255,255) и др.), hsl (hsl(10%,20%30%) и др.), шестнадцатеричным кодом (#d3d3d3,#f00 и др). Но это не всегда выгодно в плане занимаемого места в файле. Так, например, цвет заданный словом mediumspringgreen выгоднее записать шестнадцатеричным кодом #00fa9a.

## Минификация HTML

### Комментарии, пробелы, переходы на новую строку

В HTML не все комментарии пишутся исключительно для читаемости кода, но есть и условные комментарии для браузера Internet Explorer.

Листинг - Условный комментарий

<!--[if условие]> невидимый HTML-код <![endif]-->

<![if условие]> видимый HTML-код <![endif]>

Условные комментарии нельзя удалить, но обычные можно спокойно удалить.

При написании HTML-разметки используется большое количество пробельных символов для выравнивания кода, что естественно не имеет смысла для браузера при отображении. Но полностью эти выравнивающие символы удалять нельзя. Если между элементами есть пробельные символы достаточно оставить один пробел, чтоб не нарушить отображение разметки, так как элементы отображаются по-разному, в зависимости от того присутствуют пробельные символы или нет.

Листинг 1 – Пример HTML

<span>Блок 1</span> <span>Блок 2</span>

<br/>

<span>Блок 3</span><span>Блок 4</span>

На Рисунке 1 видно, что пробельные символы влияют на отображение элементов из листинга 1.



Рисунок 1 – Отображение элементов

### Внутрений код

Известно, что в документе HTML можно делать вставки CSS используя тег “<style></style>”, а также вставки JS используя тег “<script></script>”. Поэтому при обработке HTML документа необходимо находить данные теги и содержимое обрабатывать содержимое так же, как и при работе с файлами CSS и JS.

### Объединений JS/CSS файлов

Во время разработки HTML-верстка может включать в себя на различные модули. При отображении страницы эти модули начинают загружаться и для каждого модуля устанавливается новое соединение с сервером и это будет работать медленнее если бы эти все модули были объединены в один файл. Поэтому в HTML-файлах необходимо находить все JS/CSS и объединять их в отдельные файлы.

## Минификация JS

### Объединение и сжатие констант

Перед выполнением кода можно произвести манипуляции с константами, то есть объединить, сжать, посчитать. Листинг 1 может быть преобразован в Листинг 2.

Листинг 1 – Несжатый код

function test(a, b) {

run(a, 'my' + 'string', 600 \* 600 \* 5, 1 && 0, b && 0)

}

Листинг 2 – Сжатый код

function test(a,b){run(a,"mystring",18E5,0,b&&0)};

### Укорачивание локальных переменных

Изменение имен локальных переменных не повлияет на ход выполнения кода.

Листинг 1 – Необработанный код

function sayHi(name, message) {

alert(name +" сказал: " + message);

}

Листинг 2 – Обработанный код

function sayHi(a,b){alert(a+" сказал: "+b)};

### Объединение и удаление локальных переменных

Локальные переменные так же можно удалить если можно выстроить более выгодную по размеру цепочку вызовов.

Листинг 1 – Необработанный код

function test(nodeId) {

var elem = document.get(nodeId);

var parent = elem.parentNode;

alert( parent );

}

Листинг 2 – Обработанный код

function test(a){a=document.get(a).parentNode;alert(a)};

### Разворачивание if-конструкций, удаление недостижимого кода

В листинге 1 yнекоторые if-ветки могут быть удалены. А также могут быть удалены инструкции после ключевого слова “return”.

Листинг 1 – Необработанный код

function test(node) {

var id = node.id;

if (0) {

alert( "Текст 1" );

} else {

alert( "Текст 2" );

}

return;

alert( 1 );

}

Листинг 2 – Обработанный код

function test(a){alert("Text 2")};

### Переписывание синтаксических конструкций

Переписывание синтаксических конструкций так же может положительно сказаться на размере JS-кода.

Листинг 1 – Необработанный код

var i = 0;

while (i++ < 10) {

alert( i );

}

if (i) {

alert( i );

}

if (i) {

alert("1");

} else {

alert("2")

}

Листинг 2 – Обработанный код

for(var i=0;10>i++;)alert(i);i&&alert(i);i?alert("1"):alert("2");

### Инлайнинг функций

Если функци используется очень редко то подставив инструкции внутри ее в мето где она используется может уменьшить размер. В листинге 1 вызов вместо вызова функции подставлена ее реализация, а после произошло сокращение и удаление локальных переменных.

Листинг 1 – Необработанный код

function test(message) {

var elem = create(message);

showElement(elem);

function showElement(){

elem.show();

}

}

Листинг 2 – Обработанный код

function test(a){create(a).show()};

### Мелкие минификации

Имеет место также такие минификации как:

* лишние кавычки у ключей;
* замета вызовов Array/Object.

Листинг 1 – Необработанный код

var obj = {"prop" : "val" };

a = new Array();

o = new Object();

Листинг 2 – Обработанный код

var obj={prop:"val"};

a=[];

o={};

## Построение сокращенных имен

Имена идентификаторов и классов могут начинаться только с латинских букв строчных либо прописных, а последующие символы могу содержать цифр. Так же имена с прописными буквами и строчными считаются, как два разных имени. Следовательно, генерация имен будет заключаться в последовательном переборе символов. При переборе всех символов в k-ой позиции будет произведен переход к следующему символу в k+1-ой позиции, а в текущей позиции переход к первому символу. На первую позицию налаживается ограничение, оговоренное выше.

Пусть имеется словарь, сопоставляющий некоторому числу символ.

Пусть имеется массив, содержащий набор индексов символов, представляющий текущее имя.

Алгоритм:

1. увеличить первый индекс символа на 1;
2. k=0, где k – номер текущего индекса в массиве;
3. проверить на выход k-ого индекса за границу доступных символов, если выход есть, то перейти к Шагу 4, иначе к Шагу 7;
4. обнулить k-ый индекс;
5. увеличить на единицу значение k;
6. выполнить Шаг 3;
7. преобразовать текущий массив в строчное представление используя ранее описанный словарь символов.

Выполняя данный алгоритм на одном и том же массиве можно получать не повторяющиеся имена с минимальной длинной.

## Поиска имен идентификаторов и классов

### HTML

В HTML идентификаторы хранятся в значении атрибута “id”, а имена классов в значении атрибута “class”. Имена классов записаны через пробел, поэтому их легко получить, разделив строку по пробельному символу.

### CSS

В CSS идентификаторы могут записывать в селекторах. А записываются в виде строки, где первый символ это “#”, а остальная часть, это название идентификатор. С классами аналогичная ситуация только первым символом является “.”.

### JavaScript

В JavaScript есть множество способов обращения к DOM объектам страницы. Примеры функций:

* getElementById(name:string) – получение узла с идентификатором name.
* getElementsByClassName(names:string) – получение списка узлов с именами классов names, где в names имена записаны через пробел.

Так как очень сложно отследить какая именно строка была передана в данные функции, было принято решение сокращать литеральные строки, полностью совпадающие с именами ранее сокращенных имен. В большинстве случаев разработчик передает данные в эти функции именно литералом.

Так же в JS некоторые разработчики пишут собственные функции обращения к DOM элементам HTML. И часто в эти функции передаются строки символом ‘#’ перед именем идентификатора или ‘.’ перед именем класса. Описанный случай в данной курсовой работе рассматриваться не будет, но будет учтен в дальнейшей разработке.

## ASP.NET CORE - краткие сведения

ASP.NET Core - это бесплатный веб-фреймворк с открытым исходным кодом и следующее поколение ASP.NET, разработано Microsoft. Это модульная платформа, которая работает как на платформе .NET Framework, в Windows, так и на кроссплатформенном .NET Core.

Фреймворк представляет собой продукт, объединяющий ранее ASP.NET MVC и ASP.NET Web API в единую модель программирования.

Несмотря на то, что это новая структура, она имеет высокую степень совместимости концепции с ASP.NET MVC. Приложения поддерживают параллельное управление версиями, при котором различные приложения, запущенные на одном компьютере, могут настраиваться отдельно. Это невозможно с предыдущими версиями ASP.NET.

## Выводы по главе

В этой главе были рассмотрены методы минификации JS/CSS/HTML- файлов, основы парсинга, AST (abstract syntax tree), обход AST, алгоритм замены имен идентификаторов, классов.

# Практическая часть

Для решения поставленных задач был выбран язык программирования C#, а средой разработки Visual Studio.

## Antlr в Visual Studio

Так как средой разработки была выбрана Visual Studio необходимо автоматизировать процесс генерации классов лексера и парсера при изменении грамматики. Данную возможность предлагает Visual Studio. Для работы с Antlr достаточно установить через менеджер управлением пакетов NuGet пакет “Antlr4”. Это возможно сделать через графический интерфейс NuGet или выполнить команду из листинга 2.1.1.

Листинг 1 – Команда для установки пакетов Antlr

Install-Package Antlr4

Так же рекомендуется для создания и использования сгенерированных грамматик создавать отдельный проект, так как при сборке проекта все грамматики будут пересобираться.

Листинг 2 – Добавление запуска сборки файлов грамматик

<Antlr4 Include="file.g4">

<Generator>MSBuild:Compile</Generator>

<CustomToolNamespace>CustomNamespace </CustomToolNamespace>

<Listener>False</Listener>

<Visitor>True</Visitor>

</Antlr4>

После добавления в файл .csproj проекта кода из Листинга 2, Visual Studio будет используя установленный пакет собирать файл в классы реализующих лексический анализ, это файл \*Lexer, и синтаксический анализ, это файл \*Parser, где \* это название файлов, описывающих граматику. Данная компиляция будет происходить автоматически при компилировании проекта.

## Пример использования Antlr

В Листинге 2.3.1 показан пример использования фреймворка Antlr, используя сгенерированные лексер и парсер для языка JavaScript. Код закомментирован и не требует дополнительных разьяснений.

Листинг 2.3.1 – Пример использования Antlr

//получение содержимое файла

string input = File.ReadAllText(fileName);

//получения потока символов

AntlrInputStream inputStream = new AntlrInputStream(input);

//разбор потока символов на лексемы

var lexerJs = new JsLexer(inputStream);

//получение потока токенов

CommonTokenStream commonTokenStream;

commonTokenStream = new CommonTokenStream(lexerJs);

//составление из лексем AST

var parserJs = new JsParser(commonTokenStream);

## Пример использования Visitor для AST

Для обхода AST используются Visitor и Listener. Было принято решение выбрать для обхода именно Visitor так как нет необходимости одновременно отслеживать посещения в каждый узел. В листинге 1 приведен пример реализации Visitor обхода дерева HTML-файла для поиска имен идентификаторов. А в листинге 2 использование реализованного Visitor.

Листинг 1 – Пример класса Visitor

class Visitor : HtmlParserBaseVisitor<object> {

private FreqList<string> freqList;

public Visitor(FreqList<string> freqList) {

this.freqList = freqList;

}

public override object VisitHtmlAttribute(

[NotNull] HtmlParser.HtmlAttributeContext context) {

var contextName = context.htmlAttributeName();

var name = contextName.GetText();

bool isId = name.ToUpper() == “ID”

if (isId) {

var mangerAttr = new HtmlAttributeManager(context);

freqList.Increment(mangerAttr.Value);

}

return null;

}

}

Листинг 2 – Использование класса Visitor

//получения корня AST

var root = parserHtml.htmlDocument();

var freqList = new FreqList<string>()

//Создание объект класса Visitor который

//наследуется от HtmlParserBaseVisitor

var visitor = new Visitor(freqList);

//запуск обхода всего дерева

visitor.Visit(root);

## Использование Google Closure Compile

Одна из самых известных библиотек для минификации CSS-фалов. Однако она реализованно толька на языках программирования как JS и Java. Но кампания Google продумала данный вопрос и предоставляет возможность минифицировать JS-файлы отправляя их им на сервер с определенными параметрами. В листинге 1 приведен пример использования данной технологии.

Листинг 1 – Пример использования Google Closure Compile

public Stream ToOptimize(Stream stream) {

HttpWebRequest req = WebRequest

.Create("https://closure-compiler.appspot.com/compile")

as HttpWebRequest;

req.Method = "POST";

req.ContentType = "application/x-www-form-urlencoded";

var text = new StreamReader(stream).ReadToEnd();

using (var s = req.GetRequestStream()) {

var sw = new StreamWriter(s);

sw.Write(@"compilation\_level=ADVANCED\_OPTIMIZATIONS

&output\_format=text

&output\_info=compiled\_code

&js\_code=");

sw.Write(text);

sw.Flush();

s.Close();

}

var res = req.GetResponse();

return res.GetResponseStream();

}

В данно листинге открывается соединение, отправляются данные из потока на сервер с передачей параметров, и получение данных с сервера.

## Пример использования CSSO

CSSO – это библиотека для минификации CSS-файлов показывающий на данный момент самые высокие показатели. Данная библиотека реализованна на JS. Но для языка C# существует множество реализаций интерпретаторов JS-кода. Выбор был сделан в сторону Jint, в связи с его простым использованием. В Листинге 1 приведен пример использования Jint для минификации CSS-кода.

Листинг 1 – Пример использования Jint и CSSO

Engine engine = new Engine();

engine.Execute(csso);

engine.SetValue("text", text);

engine.Execute("csso.minify(text).css;")

var result = engine.GetCompletionValue().AsString();

В Листинге 1 создается интерпретатор JS, инициализируется библиотека CSSO, в переменную text записывается оптимизируемый CSS-код, далее вызывается функция библиотеки CSSO, и возвращается полученный результат.

## Пример использования Magick .NET

Magick .NET – библиотека для всевозможных манипуляций с изображениями. Но для поставленных задач необходима только одна – это оптимизация изображений. В Листинге 1 приведен пример использования данной библиотеки для минимизации изображений.

Листинг 1 – Пример использования Magick .NET

ImageOptimizer optimizer = new ImageOptimizer();

optimizer.OptimalCompression = true;

optimizer.LosslessCompress(stream);

В данном листинге проиходит инициализация объекта для минимизации изображения, установка настроек минимизации, сама минимизация потока.

## Пример использования CharpCompress

Для получения от пользователя HTML-верстки было принято решение упаковывать ее в архив для более удобной загрузки множества файлов. Библиотекой для архивации была выбрана “CharpCompress”, так как она поддерживает все популярные типы архивов. В листингах 1 и 2 приведены примеры кода для упаковки и распаковки архива соответственно.

Листинг 1 – Упаковка архива

//инициализация архива

var zip = ArchiveFactory.Create(ArchiveType.Zip);

//добавление в архив файлов

foreach (var item in files) {

zip.AddEntry(item.Key, item.Value, false);

}

//сохранение полученного архива в поток

var result = new MemoryStream();

zip.SaveTo(result, CompressionType.Deflate);

Листинг 2 – Распаковка архива

// открытие архива

var archive = ArchiveFactory.Open(stream);

//получение всех файлов и сохранение в словарь

foreach (var entry in archive.Entries.Where(e => !e.IsDirectory)) {

var ms = new MemoryStream();

entry.WriteTo(ms);

ms.Position = 0;

files.Add(entry.Key, ms);

}

## Обзор взаимодействия с разработанной библиотеки

На рисунке 1 представлена диаграмма классов с которой может взаимодействовать пользователь.

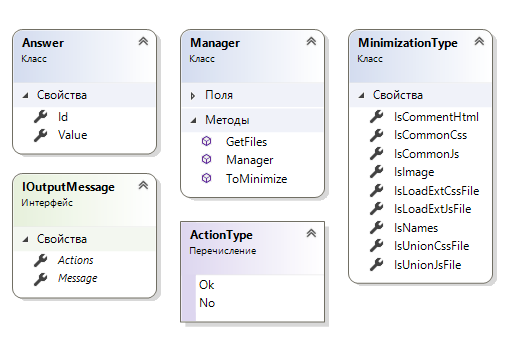


Рисунок 1- Диаграмма классов

MinimizationType – класс для передачи в библиотеку опций минификаций.

IOutputMessage – класс для принятия от библиотеки сообщений об минификации с возможностью ответить на них. Свойство “Message” возвращает сообщение. Свойство “Actions” возвращает массив доступных действий, которые может отправить пользователь библиотеке.

ActionType – перечисление возможных действий пользователя.

Answer – класс для передачи ответа пользователя классу “Manager”.

Manager – основной класс с помощью которого предоставляется возможность минимизировать класс. В конструктор данного класса передается “Dictionary<String,Stream>”, где ключ это относительное имя файла, а значение это поток, с помощью которого можно получить данные файла. Метод “ToMinimize” принимает объект класса “MinimizationType” и выполняет минификацию на основании этого объекта. Возвращает данный метод массив объектов “IOutputMessage”. Метод “GetFiles” принимает массив объектов “Answer”, если массив не пустой то производит запланированный дейтсвия. Возвращает аналогичный предыдущему объект “Dictionary<String,Stream>”.

## Пример использования разработанной библиотеки

В листинге 1 приведен пример использования разработанной библиотеки.

Листинг 1 – Пример использования бибилиотеки.

//инициализация библиотеки

Manager manager = new Manager(files);

//инициализация опций минификаций

MinimizationType min = new MinimizationType();

min.IsCommonCss = true;

min.IsCommonJs = true;

//выполнение минификаций и получение сообщений от библиотеки

var messages = manager.ToMinize(min);

//получение первого сообщения

var message = messages.First();

//инициализация ответа

Answer answer = new Answer();

answer.Id = 0;

answer.Value = ActionType.Ok;

//передача ответа библиотеки и получение минифицированных файлов

var resultFiles = manager.GetFiles(new List<Answer>{answer})

## Страница загрузки файла

Razor — это синтаксис разметки для внедрения в веб-страницы серверного кода. Синтаксис Razor состоит из разметки Razor, C# и HTML. Файлы, содержащие Razor, обычно имеют расширение CSHTML. [6]

В листинге 1 приведена форма для загрузки архива и выбора минификаций.

Листинг 1 – Форма для загрузки файла

@using Minifying.External.Models;

<form action="/UploadFile"

method="post"

enctype="multipart/form-data">

<input type="file" name="file" /><br />

@foreach(var item in typeof(MinimizationType).GetProperties()){

@Html.CheckBox(string.Concat("opt.", item.Name)) @item.Name<br />

}

<input type="submit" value="Оптимизировать" />

</form>

В листинге 1 при помощи рефлексии просматриваются все свойства класса “MinimizationType” и выводятся элементы формы “checkbox” с именами свойств. Таким образов при добавлении новых минификаций они автоматически отобразятся на данной странице.

## Выводы по главе

В этой главе содержится обзор и практическое применение используемых библиотек для решения поставленных задач, взаимодействие клиента с библиотекой, создание динамических страниц при помощи разметки Razor.

**ЗАКЛЮЧЕНИЕ**

Цель данной работы – это разработка web – приложения с графическим интерфейсом, способное автоматизировать операции, необходимые для проведения кастинга. Для достижения были решены следующие задачи: изучена предметная область, было выбрано средство реализации, определены основные сущности, был реализован графический интерфейс, в приложение добавлены операции управления сущностями, реализована интеграция с сервером.

Графический интерфейс приложения разделен на основные категории: проекты, агентства, клиенты и актеры.

Приложение предоставляет пользователю следующие возможности:

* создавать роли, на которую будет проводиться кастинг;
* создавать актеров, которые будут проходить кастинг;
* загружать материалы для ролей и актеров;
* редактировать данные проектов/агентств/клиентов/актеров;
* загружать документы для проекта;
* воспользоваться галереей для просмотра видео, фотографий;
* воспользоваться фильтрами для сортировки различных данных;
* изменять статус актера.

В ходе работы были использованы компоненты из библиотеки Angular Material, что сделало интерфейс приложения более современным и отзывчивым.

Данное приложение поддерживается в браузерах Chrome, Firefox, Opera, Edge, Internet Explorer 11. Оно позволяет упростить процесс сопровождения кастинга и манипуляцию данными.

Поставленная цель была достигнута и задачи, необходимые для достижения поставленной цели, были решены. Результатом проделанной работы стало Angular web-приложение.

**Список использованных источников**

1. Cascading Style Sheets [Электронный ресурс]. - Режим доступа: https://www.w3.org . Дата доступа: 25.05.2018.
2. Дэвид Макфарланд, CSS3: The Missing Manual 3-е издание. — 3-е. — Санкт-Петербург, 2014. — С. 608 — ISBN 978-5-496-00428-2
3. HTML5, CSS3 и JavaScript. Исчерпывающее руководство. — 4-e — Эксмо, —2014. — С. 528 — ISBN 978-5-699-67603-3
4. Кочуркин И. Обработка древовидных структур и унифицированное AST [Электронный ресурс]. – Режим доступа: <https://www.securitylab.ru/analytics/481706.php> . Дата доступа 26.05.2018.
5. JavaScript. Подробное руководство. — 1-е — Символ-Плюс, — 2008.—С. 992 — ISBN 978-5-93286-103-5
6. Microsoft Docs [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/razor. – Дата доступа: 28.05.2018

# Приложение А

Страница загрузки архива

**Приложение Б**

**Содержимое дискового носителя**

1. Casting.rar – исходный код проекта;
2. CastingBuild.rar – приложение;
3. Пояснительная записка Мысин.О.Ю.docx – пояснительная записка к дипломной работе;
4. Презентация Мысин.О.Ю – презентация диплома.