МИНОБРНАУКИ РОССИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Московский государственный технологический университет «СТАНКИН»

(ФГБОУ ВО «МГТУ «СТАНКИН»)

Институт
информационных
систем и технологий

Кафедра
информационных систем

Махмудов Бобурбек Нодирбекович

Тема: «**Исследование методов повышения производительности систем веб-шаблонов и разработка системы шаблонизации на их основе**»

Выпускная квалификационная работа на присвоение квалификации «бакалавр» по направлению 09.03.02 «Информационные системы и технологии»

Регистрационный № _____

Заведующий кафедрой
д.т.н., проф. **Позднеев Б.М.**

подпись

Руководитель
к.т.н., доцент **Бумарин Д.П.**

подпись

Студент **Махмудов Б.Н.**

подпись

Москва 2020 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

С момента своего изобретения, электронные вычислительные машины становились всё более сложными и одновременно более производительными, на сегодняшний день данная тенденция сохраняется. Например, согласно закону Мура, количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца, что увеличивает сложность интегральной схемы, но при этом увеличивая количество операций, которая она может совершать, то есть её производительность. Теория вычислительных алгоритмов тоже не стоит на месте, и с каждым годом на свет появляются различные публикации, описывающие способы уменьшения временных и других ресурсов, требуемых для решения популярных задач в области вычислений.

В данной выпускной квалификационной работе рассматривается проблема, связанная с относительно низкой производительностью системы веб-шаблонов. В ходе её написания будут проанализированы самые широко используемые решения, доступные как среди проприетарных, так и свободно распространяемых программных продуктов. На основе анализа будут выявлены наиболее распространённые причины низкой производительности. После чего будет изучен ряд мер которые можно использовать для устранения причин понижения быстродействия или уменьшения их эффекта. На основе предложенных мер будут спроектирована высокопроизводительная система шаблонизации, и разработаны требования к технологиям и инструментам для её реализации. С использованием выбранных технологий будет разработана система шаблонизации, которая должна адресовать ранее выявленные причины низкой производительности. Для оценки результатов разработки будет произведено тестирование разработанного приложения под нагрузкой, с целью оценки показателей быстродействия.

ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Термины и определения

Предметной областью данной выпускной квалификационной работы является применение систем веб-шаблонов для массового создания веб-документов. В работе используется ряд терминов, определение которых приведено ниже.

Система – комбинация взаимодействующих элементов, организованных для достижения одной или нескольких поставленных целей [1].

Веб-шаблон – HTML-код с готовым дизайном и верстkой, а также с дополнительной разметкой на языке шаблонизации, который используется для построения веб-документа.

Шаблонизатор – отдельное программное обеспечение или программный модуль, осуществляющий процесс генерации конечных веб-документов. Входными потоками процесса генерации являются веб-шаблон и контекст шаблонизации (данные), а роль управляющего потока выполняет формальная грамматика используемого языка шаблонизации.

Контекст шаблонизации – структура данных, которая содержит переменные окружения и методы, которые в свою очередь могут быть использованы шаблонизатором в процессе построения веб-документа.

Система веб-шаблонов – система, в состав которой входят три элемента:

1. Веб-шаблон;
2. Источника данных (JSON, XML, база данных);
3. Шаблонизатор.

Шаблонизатор комбинирует данные и веб-шаблоны для массовой генерации веб-документов.

Лексический анализатор (англ. Lexical Analyzer) – отдельное программное обеспечение или программный модуль, предназначенный для аналитического разбора входной последовательности символов на распознанные группы – лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах). В простых случаях понятия «лексема» и «токен» идентичны, но более сложные токенизаторы дополнительно классифицируют лексемы по различным типам («идентификатор», «оператор», «часть речи» и т. п.).

Синтаксический анализатор (англ. Parser) – отдельное программное обеспечение или программный модуль, предназначенный для сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализатором.

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, внутренние вершины которого поставлены в соответствие (помечены) с операторами языка программирования, а листья — с операндами [3].

Компиляция – сборка программы, включающая трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования высокого уровня и/или языке ассемблера, в эквивалентные программные модули на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера) или непосредственно на машинном языке или ином двоично-кодовом низкоуровневом командном языке и последующую сборку исполняемой машинной программы.

Интерпретация – процесс построчного анализа, обработки и выполнения исходного кода программы или запроса (в отличие от компиляции, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения).

Среда выполнения (англ. execution environment) – вычислительное окружение, необходимое для выполнения компьютерной программы и доступное во время выполнения компьютерной программы. В среде выполнения, как правило, невозможно изменение исходного текста программы, но может наличествовать доступ к переменным окружения операционной системы, таблицам объектов и модулей разделяемых библиотек.

Сборка мусора (англ. garbage collection) в программировании – одна из форм автоматического управления памятью. Специальный процесс, называемый «сборщик мусора», периодически освобождает память, удаляя объекты, которые уже не будут востребованы приложениями.

Системный вызов (англ. system call) в программировании и вычислительной технике – обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции, от имени этой программы.

Переключение контекста (англ. context switch) — в многозадачных ОС и средах - процесс прекращения выполнения процессором одной задачи (процесса, потока, нити) с сохранением всей необходимой информации и состояния, необходимых для последующего продолжения с прерванного места, и восстановления или загрузки состояния задачи, к выполнению которой переходит процессор.

Асинхронный ввод/вывод – является формой неблокирующей обработки ввода/вывода, который позволяет процессу продолжить выполнение, не дожидаясь окончания передачи данных. Входные и выходные

(I/O) операции на компьютере могут быть весьма медленными, по сравнению с обработкой данных. Устройство ввода/вывода может быть на несколько порядков медленнее, чем оперативная память. Например, во время дисковой операции, которой требуется десять миллисекунд для выполнения, процессор, который работает на частоте один гигагерц, может выполнить десять миллионов циклов команд обработки [15].

Зеленые потоки – это нити выполнения, управление которыми вместо операционной системы выполняет среда выполнения. Они эмулируют многопоточную среду, не полагаясь на возможности ОС по реализации легковесных потоков. Управление ими происходит в пользовательском пространстве, а не пространстве ядра, что позволяет им работать в условиях отсутствия поддержки встроенных потоков, и избегать «дорогостоящих» смен контекста.

REST (от англ. Representational State Transfer – «передача состояния представления») – архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. Для веб-служб, построенных с учётом REST (то есть не нарушающих накладываемых им ограничений), применяют термин «RESTful» [14].

1.2. Общий принцип работы систем веб-шаблонов

Все существующие системы веб-шаблонов функционируют похожим образом. Для работы системы необходимо присутствие следующих компонент:

- Шаблонизатора – основной компонент системы, который генерирует конечный документ.

- Веб-шаблонов, размеченных языком шаблонизации, синтаксис которого поддерживается используемым шаблонизатором.
- Источника данных, при этом структура данных должна соответствовать правилам формирования контекста, который может быть обработан шаблонизатором [16].

При соблюдении вышеперечисленных условий, работу системы веб-шаблонов можно визуализировать на рис. 1.2.1.

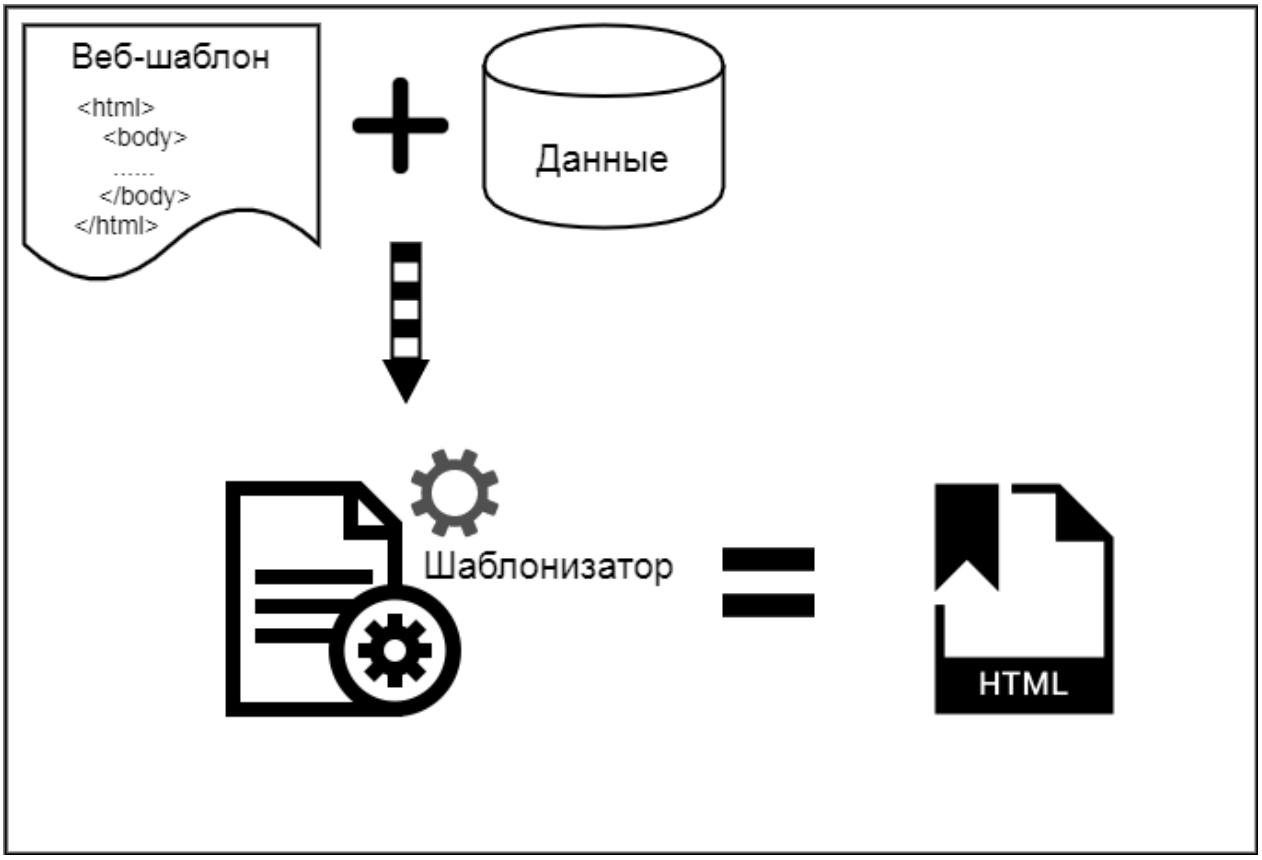

Рис. 1.2.1 Визуализация работы системы веб-шаблонов

Процесс шаблонизации в общем случае организован в последовательный набор операций, которые можно сгруппировать в четыре этапа:

1. Шаблон загружается в оперативную память. Источником шаблона может быть поток байт, файл, запись в базе данных и т.п.

2. Специальная подпрограмма, предназначенная для лексического анализа, именуемая «лексер», проводит анализ шаблона, и разбивает его на лексические единицы – лексемы, которые также называют токенами. Результатом этого процесса является поток токенов, которые значительно проще обрабатывать. Пример лексического разбора продемонстрирован на рис. 1.2.2.

3. Другая подпрограмма, целью которой является синтаксический разбор потока токенов, именуемая «парсер», производит преобразование потока токенов в особую древовидную структуру данных, известной как абстрактное синтаксическое дерево.

4. На основе абстрактного синтаксического дерева и переданного контекста шаблонизации, шаблонизатор производит построение конечного веб-документа [3]. Пример построения абстрактного синтаксического дерева показан на рис. 1.2.3.

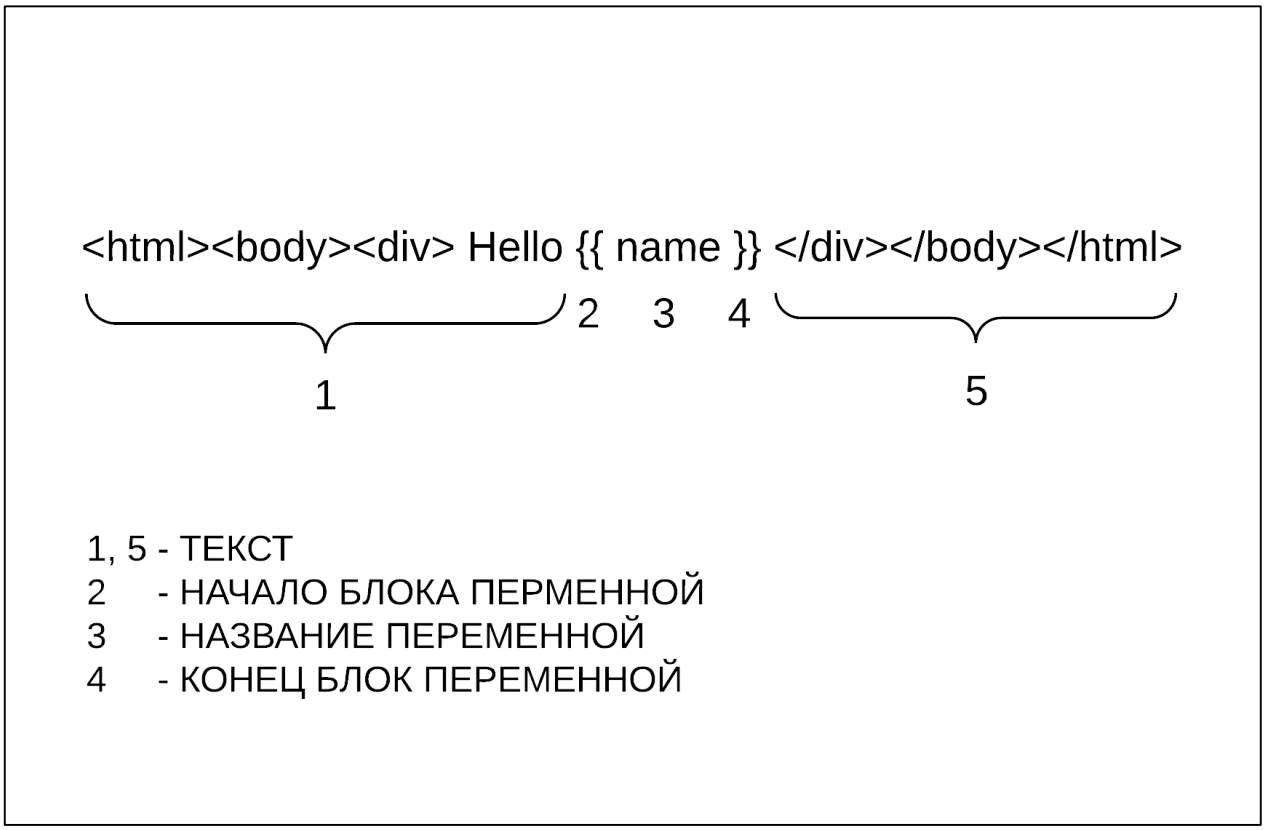

Рис. 1.2.2 Лексический разбор шаблона, каждая лексема пронумерована

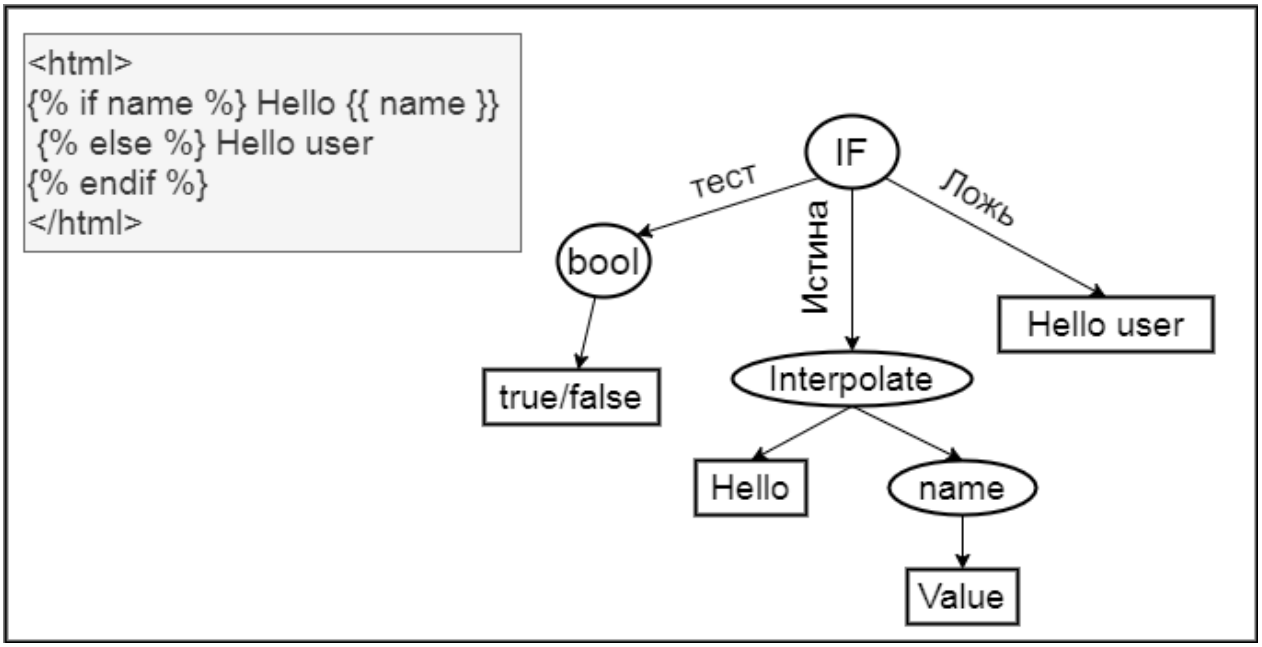

Рис. 1.2.3 Построение абстрактного синтаксического дерева

Далее можно ознакомится с функциональной моделью вышеописанного процесса в нотации IDEF0. На рис. 1.2.4 представлена контекстная диаграмма процесса шаблонизации.

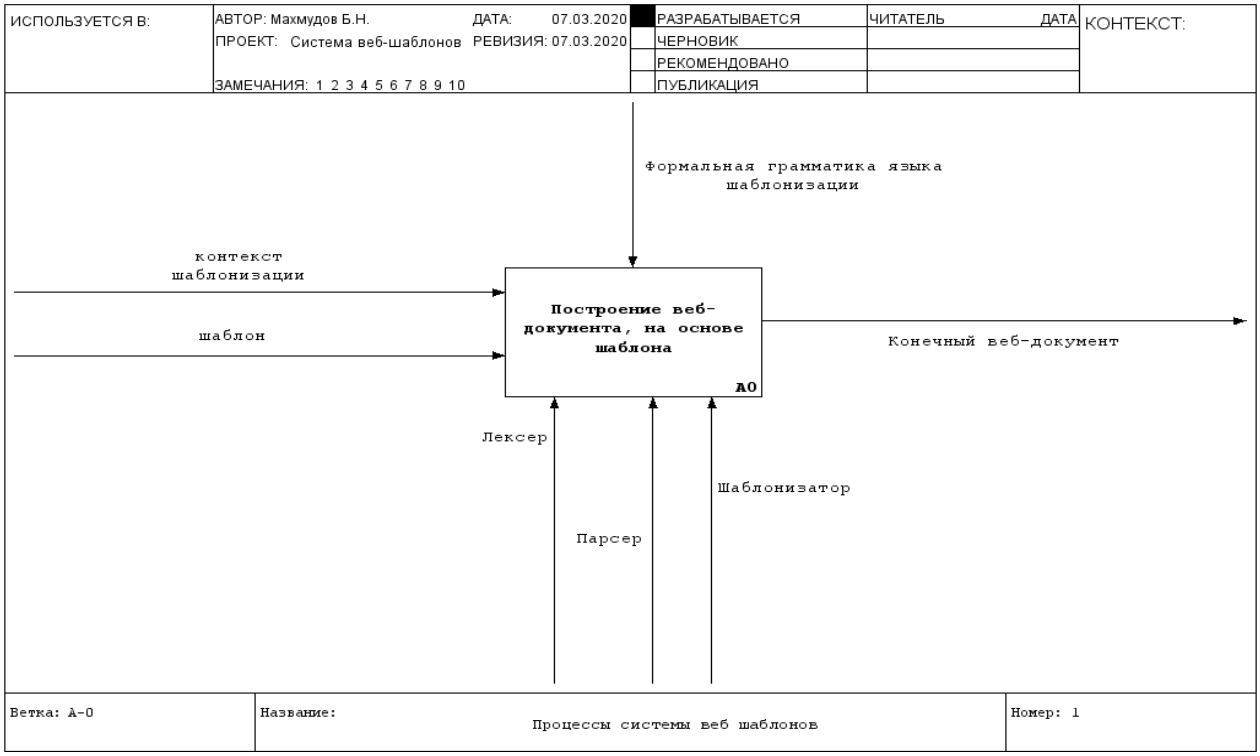

Рис. 1.2.4 Контекстная диаграмма процесса шаблонизации

На Рис. 1.2.5 приведена декомпозиция верхнего уровня процесса.

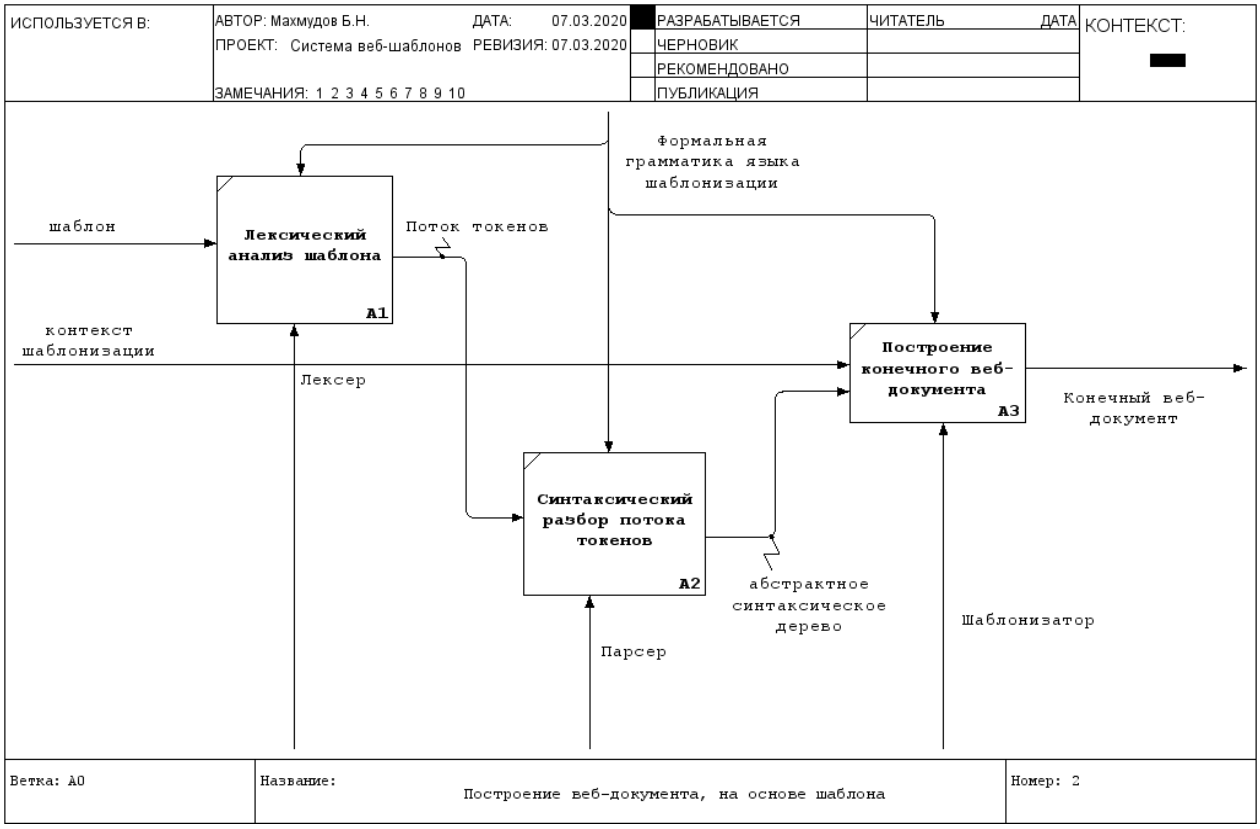

Рис. 1.2.5 Декомпозиция верхнего уровня процесса шаблонизации

1.3. Обзор существующих решений

На сегодняшний день существует большое множество систем веб-шаблонов, как проприетарных, так и с открытым исходным кодом.

Системы веб шаблонов можно выделить в две основные категории, в зависимости от способа их размещения относительно конечного пользователя:

- server-side – работающие на стороне сервера,
- client-side – работающие в составе клиентского приложения.

Первые системы веб шаблонов работали на стороне сервера, и представляли собой препроцессоры текста. Распространённой практикой было использование технологии CGI (Common Gateway Interface), которая подразумевает вызов веб-сервером внешнего программного кода, как правило скрипта, задачей которого была генерация веб-документа с использование

данных, полученных в запросе от клиентского приложения. Со временем появились полноценные веб-фреймворки, которые имеют в своём составе систему веб-шаблонов в качестве отдельного программного модуля. Далее в таблице 1.3.1 приведен перечень, наиболее популярных систем веб-шаблонов, работающих на стороне сервера.

Система веб-шаблонов	Фреймворк/Язык	Описание
Blade	Laravel/PHP	Также, как и сам фреймворк, частью которого является, Blade обладает свободной лицензией и открытым исходным кодом.
Django	Djanjo/Python	обладает широкими возможностями шаблонизации. На основе синтаксиса языка шаблонизации, используемого в Django было создано большое количество других систем веб-шаблонов. Обладает свободной лицензией.
Mustache	Различные языки	скорее спецификация по созданию систем веб-шаблонов, нежели отдельно взятый программный продукт, но существует не мало имплементаций данной спецификации на разных языках программирования. Синтаксис, определенный в спецификации, является одним из самых распространённых и широко-используемых.
Twig	PHP	система веб-шаблонов, синтаксис которой основан на Django. Наиболее

		часто используется при проектировании веб-приложений по паттерну MVC (Model View Controller). Свободная лицензия.
Lasso	Lasso/C	является сервером приложений, имеющий в своем составе одноимённую подсистему веб-шаблонов. Является примером проприетарной систем веб-шаблонов. По возможностям значительно уступает свободным аналогам

Таблица 1.3.1 Системы веб-шаблонов исполняемые на стороне сервера

Также существуют системы веб-шаблонов, которые могут исполнятся прямо в клиентском приложении, как правило это веб-браузер. Основной принцип работы этой категории систем веб-шаблонов состоит в том, что они взаимодействуют с объектной моделью документа [5] (DOM), с целью изменения его внешнего представления. Достигается это путём использования сценарных языков, наиболее популярным из которых является JavaScript. Ниже приведён таблица систем веб-шаблонов, исполняемых на стороне клиента.

Система веб-шаблонов	Фреймворк/Язык	Описание
Handlebars	JavaScript	синтаксис этой системы веб-шаблонов вдохновлён спецификацией Mustache, и имеет с ней полную обратную совместимость. Для ускорения генерации документа, Handlebars заблаговременно компилирует веб-шаблон в JavaScript код, что положительно сказывается на

		его производительности.
Dot.js	JavaScript	является одним из самых быстрых и лаконичным из всех существующих шаблонизаторов на языке JavaScript. Но из-за своей легковесности, не обладает всеми возможностями аналогов.
Squirrelly	JavaScript	легковесная встраиваемая система веб-шаблонов, разработанная в качестве библиотеки шаблонизации, которая обладает высокими показателями эффективности. Не смотря на свой маленький размер, Squirrelly предоставляет широкий спектр возможностей шаблонизации

Таблица 1.3.2 Системы веб-шаблонов исполняемые на стороне клиента

Все приведённые выше системы веб-шаблонов были разработаны для решения определённых задач, и каждая из них обладает рядом достоинств и недостатков. Выбор той или иной системы веб-шаблонов, да и любого программного средства в целом, как правило, обусловлен несколькими факторами, среди которых можно выделить:

- Уровень навыков и опыт команды разработчиков в применении определённой технологии. Разработчики часто отдают свои предпочтения уже известным и проверенным решениям, так как внедрение новой технологии всегда подразумевает определённые риски, связанные с неопределённостью.
- Окружение. А именно какой перечень технологий уже используется, и как хорошо новое программное средство будет взаимодействовать с существующим окружением.

- Удобство в использовании. Насколько продукт понимаемый, изучаемый, используемый и привлекательный для пользователя в заданных условиях [2]. Иначе говоря, пользователь должен иметь возможность эксплуатировать программное средство в определенных условиях для достижения установленных целей с необходимой результативностью, эффективностью и удовлетворённостью.
- Удобство в сопровождении. Сколько усилий нужно прилагать для доработок и поддержки программного продукта в соответствии с изменяющимися требования заказчиков.
- Производительность. Насколько быстро работает программное обеспечение под определённой нагрузкой. Часто пренебрегаемый критерий выбора, так как на практике его перевешивают предыдущие пункты. Это связано с тем что быстродействие системы веб-шаблонов не является «узким местом» для большинства надсистем в которых она функционирует, и время, затрачиваемое на выполнение ею полезной работы, находится в «приемлемых» пределах.
- Эффективность. Насколько достигнутые результаты соотносятся с затраченными ресурсами. Немаловажный фактор, в связи с тем, что параллельно с системой веб-шаблонов могут исполнятся другие программные средства, которые также потребляют системные ресурсы.

Именно производительность и эффективность, и являются центральной темой исследования данной выпускной квалификационной работы.

1.4. Основные причины низкой производительности шаблонизаторов

В разделе «Обзор существующих решений» был приведён перечень самых широко-используемых систем веб шаблонов, проанализировав эти решения, можно сделать вывод о том, что большинство из них имеет ряд

общих признаков, тем или иным образом влияющие на производительность системы в целом.

1. Они имеют в своем составе шаблонизатор, написанный на интерпретируемом языке.
2. Имплементация шаблонизатора, часто использует выделение памяти из управляемой кучи.
3. Наличие сборщика мусора, входящее в состав среды исполнения.
4. Необходимость заново проводить шаги загрузки, лексического и синтаксического анализов при построение нового шаблона.
5. Простой ресурсов многоядерных процессоров.

Написание шаблонизаторов на интерпретируемых языках, пожалуй, является самым значительным из факторов, вносящих вклад в снижение производительности и эффективности функционирования систем веб-шаблонов. Программный код, написанный на интерпретируемых языках значительно уступает компилируемым аналогам в быстродействии, в связи с тем, что в ходе выполнения процесса интерпретации, присутствует дополнительный шаг. Дополнительный шаг предполагает необходимость построчной трансляции написанного кода в промежуточное представление, и его последующей интерпретации в машинный код. Из этого следует потребность в достаточно сложной и большой среде выполнения, что требует дополнительных пространственных и временных ресурсов вычислительной системы. Также тот факт, что трансляция происходит построчно, прямо во время исполнения программного кода, делает невозможным применение достаточно большого количества оптимизаций, который могут производить компиляторы, связанные с уменьшением времени выполнения программы, размера исполняемого файла, или энергопотребления в ходе её исполнения. Наглядно разницу между процессами компиляции и интерпретации можно увидеть на рис. 1.4.1. Достоинством таких языков, является наличие

различного рода высокоуровневых абстракций, облегчающих процесс кодирования, а также простота в отладке и поддержке программного кода. Они идеально подходят для разработок по модели быстрого прототипирования.

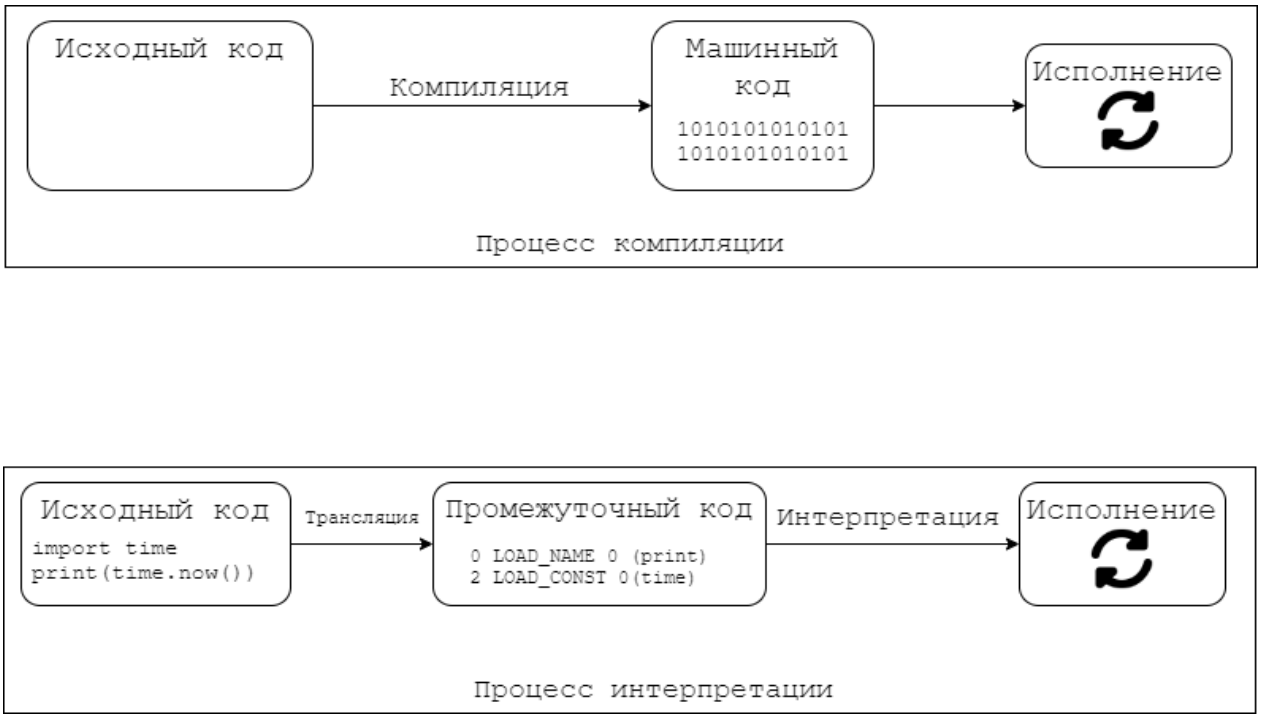

Рис. 1.4.1 Сравнение процессов компиляции и интерпретации

Частое выделение памяти из управляемой кучи, и копирование данных из одной области памяти в другую, является ещё одним немаловажным фактором оказывающее существенное влияние на снижение быстродействия. Это часто требуемая операция при работе со строками, например, их соединения. Также если памяти в управляемой куче недостаточно, то будет использован системный вызов, для расширения кучи. Использование системного вызова, всегда подразумевает смену контекста исполнения, то есть управление процессором переходит операционной системе. Смена контекста, сам по себе является «дорогой» операцией, так как представляет собой сохранение состояния исполняемой программы, с последующим восстановлением этого состояния. Во время процесса сохранения и загрузки

состояния потоков исполнения, не выполняется никакой полезной работы, соответственно это оказывает негативное влияние на общее время выполнения программы. Также использование системного вызова может возникнуть в случайный момент времени и продолжаться случайное время. С графической схемой смены контекста можно ознакомится на рис. 1.4.2.

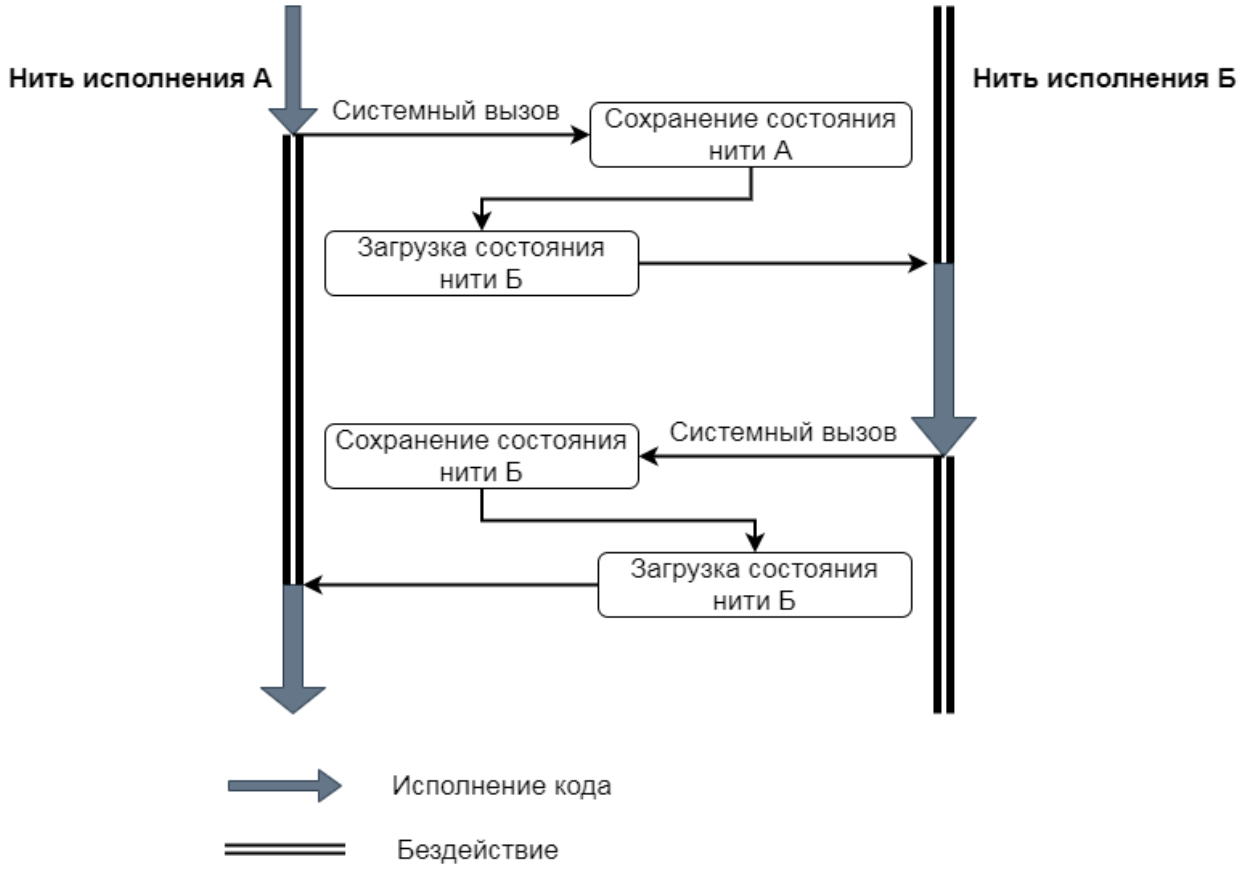

Рис. 1.4.2 Графическая схема смены контекста

Сборщик мусора является неотъемлемой частью многих языков программирования, как интерпретируемых, так и компилируемых. Его наличие значительно снижает, а иногда и вовсе отменяет необходимость в ручном управлении памятью, а именно её выделение и последующее освобождение. Он входит в состав среды выполнения программного обеспечения, и хранит сведения обо всей памяти которая в данный момент

используется в программе, часто это делается с помощью подсчёта ссылок на конкретную область или «объект». Регулярно сборщик мусора сканирует управляемую кучу на наличие «осиротевших» областей памяти, и освобождает их. Ещё одной задачей сборщика мусора может являться дефрагментация управляемой кучи, при невозможности выделения из неё памяти. Все эти процессы происходят в фоновом режиме, и не требуют вмешательств со стороны разработчика. Но как сканирование, так и дефрагментация в особенности, являются очень затратными по отношению к системным ресурсам операциями, в следствии чего производительность шаблонизатора существенно деградирует. Принцип работы, основанный на подсчёте активных ссылок на объекты в управляемой куче проиллюстрирован на рис. 1.4.3.

Большинство систем веб-шаблонов не имеют или не используют по умолчанию механизм кеширования уже скомпилированного веб-шаблона, это в свою очередь может привести к ситуациям, где все три шага шаблонизации, предшествующие шагу построения документа приходится проводить каждый раз, когда меняется текущий шаблон. Если система веб-шаблонов используется в среде, где смена шаблона происходит с высокой частотой, то необходимость проводить загрузку, лексический и синтаксические анализы шаблона большое количество итераций, может привести к критически низким показателям производительности.

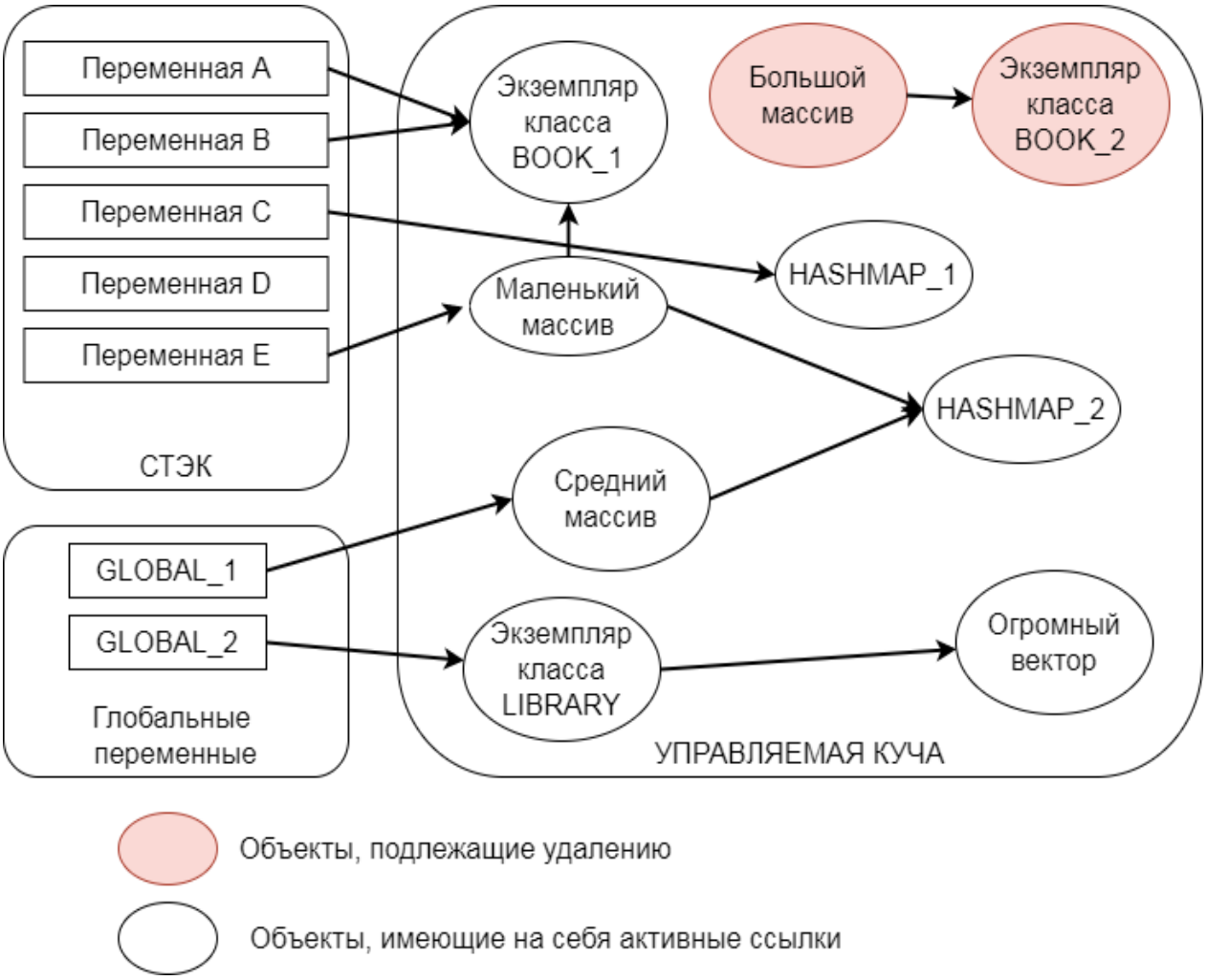

Рис. 1.4.3 Принцип работы «сборщика мусора»

Простой ресурсов многоядерных процессоров – это проблема, возникающая ввиду сильной связанности шагов процесса шаблонизации и невозможностью разделения данной задачи на подзадачи между несколькими нитями исполнения. В вычислительных системах с несколькими центральными процессорами, и отсутствием иного рода вычислительных задач, это может привести к простою или бездействию остальных процессоров, при этом время работы самого шаблонизатора может увеличиться почти пропорционально количеству бездействующих вычислительных единиц.

Приняв во внимание вышеперечисленные проблемы и их специфику можно сформировать перечень целей и задач, которые необходимо достичь и решить соответственно, в рамках данной выпускной квалификационной работы.

1.5. Цели и задачи

Как говорилось ранее, в данной работе основное внимание отводится производительности и эффективности систем веб-шаблонов. В разделе «Основные причины низкой производительности шаблонизаторов» был приведён и подробно разобран перечень факторов, оказывающих негативное влияние на быстродействие системы веб-шаблонов в целом. Отталкиваясь от этой информации, можно сформулировать основные задачи, который предстоит решить в рамках этой выпускной квалификационной работы:

1. Исследовать альтернативные способы написания шаблонизаторов на компилируемых языках программирования, при этом необходимо минимизировать отрицательное воздействие такого подхода на такие показатели качества, как удобство в использовании и простота сопровождения программного продукта.

2. Изучить парадигмы и практики в программировании, которые позволяют уменьшить количество операций, связанных с выделением памяти и копированием данных между разными её областями. При этом не должно происходить значительного усложнения модели управления памятью вычислительной системы.

3. Проанализировать существующие языки программирования, которые не реализуют управление памятью через «сборку мусора». Оценить предоставляемые ими возможности, и сделать вывод в целесообразности отказа от сборщика мусора.

4. Исследовать методологии и концепции кеширования результатов уже осуществлённых вычислений, с целью минимизации накладных расходов, связанных с необходимостью повторного выполнения всех шагов шаблонизации, возникающих как следствие смены текущего веб-шаблона.

5. Ознакомится с парадигмой многопоточного программирования и изучить сложности, которые могут возникать при написании многопоточных программ. Также необходимо исследовать концепцию асинхронного программирования, оценить возможность её применения для ещё более эффективного использования системных ресурсов.

Описанные выше задачи, отражают пять основных проблем низкой производительности шаблонизаторов. В ходе решения этих задач будет собрана большая база знаний, которые в дальнейшем могут быть использованы для реализации высокопроизводительной, и в то же время удобной в использовании системы веб-шаблонов, что в свою очередь является главной целью данной выпускной квалификационной работы.

ГЛАВА 2. ИССЛЕДОВАНИЕ ВОЗМОЖНЫХ СПОСОБОВ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ СИСТЕМ ВЕБ ШАБЛОНОВ

2.1. Устранение причины низкой производительности, связанной с использованием интерпретаторов

Так как использование любых языков программирования, нуждающиеся в предварительной интерпретации для выполнения кода не допустимо для достижения повышенной производительности, то нужно искать альтернативы среди компилируемых языков программирования.

Компилируемый язык программирования – язык программирования, исходный код которого преобразуется компилятором в машинный код и записывается в файл с особым заголовком и/или расширением для последующей идентификации этого файла, как исполняемого операционной системой, в отличие от интерпретируемых языков программирования, чьи программы выполняются пошагово программой-интерпретатором.

Языки программирования принято разделять на компилируемые и интерпретируемые в силу типичных различий:

- скорость выполнения программы, скомпилированной в машинный код, превосходит скорость интерпретируемой программы, как правило, в десятки и сотни раз;
- в случае использования компилятора, при внесении изменений в исходный код программы, прежде чем эти изменения можно будет увидеть в работе программы, необходимо выполнить повторную компиляцию исходного текста.

Также среди компилируемых языков принято выделять две различных категории в зависимости от выходных данных компиляции:

1. языки, которые компилируются в машинный код целевой архитектуры вычислительной машины;

2. языки, которые компилируются в машинный код некой виртуальной машины, также известный как байт-код.

Программы, написанные на языках, компилируемые в машинный код целевой вычислительной машины очень компактны, не требуют вспомогательных средств окружения для исполнения, не зависят от исходного кода после процесса компиляции и самое главное, они обладают высокими показателями быстродействия. Их основной недостаток – это низкий уровень переносимости, из-за необходимости перекомпилировать, а возможно и адаптировать исходный код под каждую целевую архитектуру вычислительных машин. Примерами таких языков являются:

- C,
- C++,
- Haskell,
- Go,
- Rust,
- Swift.

Что же касается языков, компилируемые в байт-код виртуальной машины, то для исполнения они требуют эту самую виртуальную машину, могут выполнятся на любой архитектуре, где есть виртуальная машина. По быстродействию в разы уступают аналогам, компилируемых под конкретную архитектуру. Основной недостаток – это необходимость наличия виртуальной машины, которая также негативно влияет на производительность исполняемой программы. Примерами таких языков являются:

- Java,

- C#,
- Erlang.

Приняв во внимание вышеописанные различия среди компилируемых языков, а также тот факт, что основная задача – это повышение производительности, то выбор языка будет осуществляться в категории компилируемых в машинный код целевой архитектуры.

В целях облегчения разработки, и упрощения сопровождения проектируемой системы веб-шаблонов, выбираемый язык должен удовлетворять следующим требованиям:

- наличие зрелой экосистемы вокруг языка, то есть активное сообщество, наличие библиотек для решения часто возникающих задач;
- наличие какого-либо программного механизма для управления пакетами и зависимостями между ними;
- поддержка обобщённого программирования и какой-либо формы полиморфизма;
- наличие встроенных механизмов документирования кода.

В следующих разделах будут исследованы другие способы повышения производительности системы веб-шаблонов, и как следствие будут предъявлены дополнительные требования к другим аспектам выбираемого языка программирования.

2.2. Уменьшение операций по выделению памяти

В высокоуровневых языках программирования, частое выделение памяти из управляемой кучи, связано с абстракциями над ссылочными типами данных, которые призваны упростить процесс программирования, за счёт сокрытия от пользователя нижележащих механизмов управления памятью. Но ценой таких абстракций является лишение того уровня контроля

над памятью, которое возможно в более низкоуровневых языках, в которых ссылочные типы данных доступны. Преимущество ссылочных типов в том, что программист сам решает, когда ему нужно создать новый экземпляр некоторого объекта, выделив для этого память, и когда можно просто сослаться на уже существующий объект, посредством его адреса в памяти. Недостаток такого подхода заключается в увеличении количества ошибок в коде, которые часто бывает сложно отлаживать, такие как «утечки» памяти или двойные освобождения. Не смотря на подверженность ошибкам, использование ссылочных типов данных, вместо создания нового экземпляра объекта, при работе с большим количеством строковых данных, может положительно сказаться на быстродействии.

Ещё одним способом снижения количества выделений памяти, является использование предварительного выделения достаточного для целевой задачи объема памяти, так как время, затраченное на выделение, не зависит от объема запрашиваемой памяти (при условии малой фрагментации управляемой кучи). В результате можно сэкономить на количествах запросах на выделение памяти. Это в свою очередь, подразумевает тщательный анализ вычислительной задачи, с целью определения нижних и верхних пределов объема памяти, требуемой на её выполнение, например, на основе статистических данных.

На основе этих наблюдений, можно сформулировать дополнительные требования к выбираемому языку программирования:

- наличие поддержки ссылочных типов данных;
- наличие встроенных механизмов, снижающие вероятность возникновения ошибок, связанных со ссылочными типами данных.

Среди компилируемых языков, которые предоставляют ссылочные типы данных и соответственно более высокий уровень контроля над памятью,

доступной исполняемой программе, можно выделить: C/C++, Rust, Go, Haskell.

C и C++ являются пожалуй самыми старыми из перечисленных языков, их отличительной особенностью является принцип «trust the programmer», т.е. «доверяй программисту», это подразумевает то, что язык даёт полный контроль над вычислительными ресурсами машины. Потенциально это позволяет писать очень быстрый код, с маленьким размером исполняемого файла. Но как оказывается на практике, ввиду отсутствия в данных языках каких-либо механизмов безопасности, написанные программы подвержены появлению различного рода, сложно-отслеживаемых и отлаживаемых ошибок.

Haskell – стандартизированный чистый функциональный язык программирования общего назначения. Является одним из самых распространённых языков программирования с поддержкой отложенных вычислений. В силу того, что язык поддерживает только функциональную парадигму программирования, написание на нём программ, является не лёгкой задачей для тех, кто пришёл из более традиционного процедурного или объектно-ориентированного языка программирования. Ещё одной особенностью Haskell является то, что все типы данных неизменяемые, что подразумевает создание нового объекта в памяти каждый раз, когда нужно мутировать данные.

Go – компилируемый многопоточный язык программирования, разработанный внутри компании Google. Разрабатывался как язык программирования для создания высокоэффективных программ, работающих на современных распределённых системах и многоядерных процессорах. Он может рассматриваться как попытка создать замену языкам Си и C++ с учётом изменившихся компьютерных технологий и накопленного опыта разработки крупных систем. Является хорошим кандидатом для написания

высокопроизводительных систем. Единственным недостатком, связанным с целями данной работы, является наличие сборщика мусора в среде выполнения, который будет подробно разобран в следующем разделе.

Rust – мультипарадигмальный компилируемый язык программирования общего назначения, сочетающий парадигмы функционального и процедурного программирования с объектной системой, основанной на типажах, и с управлением памятью через понятие «владения», что позволяет обходится без сборки мусора. Язык с самого начала проектировался для использования в системном программировании, где производительная эффективность является одной из самых важных показателей качества. В отличие от языков C/C++, безопасность в смысле снижения количества ошибок, заложена в сам язык на фундаментальном уровне. Язык имеет относительно высокий порог обучения.

Далее будет разобран механизм сборки мусора, и возможные решения, которые позволяют обходится без него, с минимальными потерями в плане продуктивности и удобства в использовании.

2.3. Альтернатива сборщику мусора

Как было сказано ранее, сборка мусора, является неотъемлемой частью сред выполнения многих языков программирования. Но за удобство и простоту использования, которую предоставляет сборщик мусора, нередко приходится платить производительностью и высокими уровнями потребления памяти вычислительной системы. Это связано с тем, что для правильной работы, сборщик мусора ставит исполнение программы на «паузу», после чего выполняет освобождение памяти от неиспользуемых объектов, хотя длительность таких остановок невелика и современные сборщики мусора реализуют, оптимизированные на скорость выполнения, алгоритмы сканирования памяти на неиспользуемые объекты, тем не менее это не может

не оказывать негативного воздействия на быстродействие всей программы в целом.

Существует несколько альтернативных методик программирования, которые позволяют обходится без сборки мусора вовсе, далее приведены наиболее часто применяемые из них.

Ручное управление памятью – самый простой и прямой способ, программист сам решает, когда ему нужно выделение памяти, а после её получения берёт на себя ответственность за её своевременное освобождение. Идеально подходит для небольших и несложных программ. Но с ростом числа строк кода, и сложности программной логики, такой подход к управлению памятью, ведёт к росту числа ошибок, некоторые из которых может быть достаточно сложно отладить.

Создание пула объектов – подход который использует методику предварительного выделения памяти для различных типов данных, используемых в ходе исполнения программы. Объекты создаются заранее в этой области памяти, и предназначены для многократного использования, что позволяет обходится без дальнейших выделений памяти. Достаточно сложный в реализации, так как при изменении типов данных программы, необходимо эти изменения учитывать и в механизме пула объектов.

Использование стека вместо управляемой кучи – суть подхода заключается в том, что если размерность всех типов данных известна заранее, то можно обойтись без выделения памяти вовсе, и создавать все объекты прямо на стеке, что является гораздо более быстрой операцией, и снимает с программиста ответственность за освобождение памяти, так как после завершения функции весь стек будет автоматически освобожден. Данный подход имеет крайне узкое применение, ввиду того, что на практике очень часто приходится работать с типами данных, размерность которых определяется во время выполнения программы. Но его можно использовать в

связке с другими подходами по управлению памятью, в силу описанных выше преимуществ.

Получение ресурса есть инициализация (англ. Resource Acquisition Is Initialization (RAII)) – программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение – с уничтожением объекта. В первые идиома была реализована на языке C++, она требует от программиста тщательного проектирования классов объектов, так чтобы при окончании времени жизни переменной, «владеющей» экземпляром класса, этот экземпляр класса также был уничтожен. Идиома сложна в реализации, но при правильном использовании, сильно упрощает написание кода, без отрицательных эффектов на производительность программы.

Проанализировав альтернативные подходы к управлению памятью, можно дополнить перечень требований к выбираемому языку программирования:

- отсутствие механизма сборки мусора, или наличие возможности её отключения;
- наличие программных механизмов, реализации идиомы RAII для управления памятью.

В данном разделе были изучены существующие способы управления памятью без сборщика мусора. Можно сделать вывод о том, написание программ без использования сборки мусора, не только возможно, но и может положительно сказаться на производительности и эффективности, разрабатываемого программного обеспечения.

2.4. Кеширование наиболее часто используемых шаблонов

Необходимость в кешировании часто возникает в самых различных задачах, связанных с разработкой программного обеспечения. Основная идея кеширования заключается в сохранении результата какой-либо вычислительной операции для дальнейшего его использования из локального хранилища «кеша». При этом происходит экономия времени за счёт отсутствия потребности в повторном выполнении этой вычислительной операции, но возрастает потреблении пространственных ресурсов. Для решения проблемы повторного выполнения всех шагов шаблонизации, можно использовать локальный для шаблонизатора «кеш», который будет содержать уже построенные абстрактные синтаксические деревья, предыдущих шаблонов. В этом случае, если поступит запрос на генерацию веб-документа на основе веб-шаблона, который ранее был уже использован, то все шаги шаблонизации предшествующие построению веб-документа, можно пропустить, так как абстрактное синтаксическое дерево уже имеется в кеше.

В результате использования кеша возникают две основные проблемы:

1. Необходимость поддерживать когерентность кеша.
2. Необходимость ограничить размер кеша.

Суть проблемы когерентности кеша, заключается в том, что данные в кеше должны соответствовать тем данным, на основе которых они были получены. Решением данной проблемы, в случае системы веб-шаблонов, является триггерное обновление кеша при изменении шаблона, на основе которого было построено абстрактное синтаксическое дерево. Обновление кеша необходимо проводить с блокировкой шаблонизатора, с целью исключить вероятность использования кеша другими нитями исполнения, во время его обновления.

Необходимость ограничить размер кеша, возникает как результат фиксированности размеров памяти, доступной для использования шаблонизатором, и потенциально неограниченным количеством шаблонов, имеющимся в системе. Для решения данной проблемы можно использовать алгоритм вытесняющего кеширования LRU (Last Recently Used – Вытеснение давно неиспользуемых). Суть алгоритма заключается в том, что заранее задается размер кеша, в случае шаблонизатора это максимальное количество абстрактных синтаксических деревьев, которые могут находится в кеше. С каждым вхождением в кеше, ассоциируется «бит возраста», который при обращении к этому вхождению становится равным нулю, а для всех остальных вхождений увеличивается на единицу. В результате, при переполнении кеша, и необходимости загрузить в него новое АСД, то вхождение с наибольшим «возрастом» вытесняется из кеша и заменяется новым.

2.5. Решение проблемы простоя системных ресурсов

Последний способ повышения производительности системы веб-шаблонов, который будет рассмотрен и применён в рамках выполнения данной выпускной квалификационной работы, это задействование простаивающих вычислительных ресурсов. Как говорилось ранее проблема простаивания системных ресурсов, в случае с шаблонизаторами, заключается в сильной связанности шагов шаблонизации и элементарных операций, из которых они состоят. Например, нельзя взять большой шаблон и проделать его лексический анализ по «кускам» в разных потоках исполнения, или нельзя начать делать синтаксический анализ не имея поток лексем, которые в свою очередь являются результатом предыдущего шага – лексического анализа. Иначе говоря, весь процесс шаблонизации должен протекать в

строго определенном порядке, и разбить этот процесс на подзадачи для параллельного выполнения, не представляется возможным.

Но, как показывает практика, шаблонизаторы часто используются для генерации большого количества веб-документов за краткие промежутки времени. В случае с однопоточной реализации, все заявки на генерацию документа будут выстраиваться в очередь, один поток исполнения будет обрабатывать заявки один за другим, а остальные вычислительные ресурсы будут простаивать в условиях отсутствия иного рода задач. Так как построение одного веб-документа никак не связано с генерацией другого, то можно производить генерацию нескольких веб-документов одновременно, используя несколько потоков исполнения. Таким образом можно «догружать» простаивающие ресурсы вычислительной системы.

Не смотря на применение многопоточного подхода для решения этой проблемы, это не исключает появления ситуаций, в результате которых, ядра процессора не будут выполнять полезную работу. Это связанно с блокирующим вводом/выводом, суть которого заключается в том, что при совершении операций по вводу или выводу, процессор будет бездействовать (будут заблокирован), по причине того, что современные процессоры значительно быстрее чем устройства ввода и вывода. В случае с шаблонизатором блокирующий вводом может быть чтение шаблона с диска, а вывод – это его обновление, то есть запись на диск. Во избежание подобных блокирующих операций, применяется асинхронный ввод/вывод. В отличие от блокирующего, асинхронный код не будет дожидаться завершения операции по вводу/выводу, а продолжит исполнение другой полезной работы, если такова имеется, или отдаст управление вычислительными ресурсами другой задаче, а среда выполнения проследит за тем чтобы, ожидающая задача была продолжена, как только ввод/вывод завершится. Для эффективного управления вычислительными ресурсами, асинхронные среды выполнения

реализуют так называемые «зеленые потоки», которые ставятся в соответствие с настоящими потоками, предоставляемые операционной системой. Каждому зеленому потоку отдаётся задача на выполнение, с выделением вычислительного ресурса, и если при выполнении задачи зеленый поток столкнётся с блокирующим вводом/выводом, то он немедленно отдаст управление вычислительным ресурсом другому ожидающему потоку.

Приняв во внимание вышеописанные методы использования всех доступных вычислительных ресурсов системы, можно добавить последние требования к выбираемому языку программирования:

- поддержка многопоточного программирования;
- поддержка асинхронного программирования.

В следующем разделе будет выбран язык программирования для написания системы веб-шаблонов, с учётом всех вышеприведённых требований.

2.6. Выбор языка программирования

На основе приведённых выше требований были отобраны несколько языков программирования, который потенциально могут быть использованы для реализации проекта системы шаблонизации. Эти языки: C, C++, Go, Rust, Haskell. Далее приведена таблица сравнения данных языков, которая демонстрирует степень, с которой тот или иной язык удовлетворяет требованиям.

Язык программирования / Требование	C	C++	Go	Rust	Haskell
Наличие зрелой экосистемы	Да	Да	Да	Да	Да

Наличие пакетного менеджера	Нет	Нет	Да	Да	Да
Поддержка обобщённого программирования	Нет	Да	Нет	Да	Да
Поддержка механизмов документирования кода	Нет	Нет	Да	Да	Да
Поддержка ссылочных типов данных	Да	Да	Нет	Да	Нет
Наличие механизмов снижающих риск возникновения ошибок, связанных с управлением памятью	Нет	Нет	Да	Да	Да
Отсутствие сборщика мусора	Да	Да	Нет	Да	Нет
Наличие механизмов реализации идиомы RAII	Да	Да	Нет	Да	Нет
Поддержка многопоточного программирования	Да	Да	Да	Да	Да
Поддержка многопоточного программирования	Да	Да	Да	Да	Да

Таблица 2.6.1 Сравнение языков программирования на предмет удовлетворения требованиям

В ходе детального анализа существующих языков программирования, изучения их преимуществ и недостатков, для разработки проекта системы веб шаблонов был выбран компилируемый язык программирования Rust. Сам язык и его возможности кратко описывались ранее. Далее повторно будет приведён перечень требований, которые были предъявлены к языку, и каким образом Rust им удовлетворяет.

Наличие зрелой экосистемы вокруг языка, то есть активное сообщество и наличие библиотек для решения часто возникающих задач. Rust является относительно молодым языком, его первый публичный релиз состоялся в 2010-ом году, но не смотря на это, на текущее время он является одним из самых бурно-развивающихся языков программирования. Rust имеет очень активное и отзывчивое сообщество, очень детализированную документацию, несколько книг, написанных членами сообщества и находящихся в свободном доступе. Сообщество активно разрабатывает библиотеки для решения самого различного рода задач, подавляющее большинство из которых доступны под свободными лицензиями.

Наличие какого-либо программного механизма для управления пакетами и зависимостями между ними. Rust имеет свой набор инструментов, предназначенных для управления пакетами и зависимостями, основной среди которых это Cargo, который поддерживает управления зависимостями из различных источников, таких как центральный репозиторий пакетов crates.io или github.

Поддержка обобщённого программирования и какой-либо формы полиморфизма. Rust имеет встроенную поддержку обобщённого программирования, посредством использования типажей. Также с помощью тех же типажей можно реализовать параметрический полиморфизм. Особенностью Rust является то, что обобщённый код будет приведён к конкретным типам данных ещё на этапе компиляции (процесс мономорфизации), а не во время выполнения, что в свою очередь положительно сказывается на быстродействии и размере исполняемой программы.

Наличие встроенных механизмов документирования кода. Документировать код в Rust можно прямо в исходных файлах. В набор инструментов разработчика входит утилита rust-doc, которая просканирует

весь проект на наличие комментариев для документации, и скомпилирует их в единый html формат, удобный для публикации.

Наличие поддержки ссылочных типов данных. Так как Rust с самого начала задумывался как язык для системного программирования, то поддержка ссылочных типов данных встроена в язык на самом базовом уровне, более того в языке поддерживаются «чистые» указатели, хотя их использование считается не безопасным.

Наличие встроенных механизмов, снижающие вероятность возникновения ошибок, связанных со ссылочными типами данных. Безопасность является одним из основных целей, которые преследовали разработчики языка, из-за чего в нём с самого начала были заложены явные правила написания кода – лучшие практики, но в явном, стандартизованном виде. При нарушении этих правил, компилятор просто откажется компилировать данный код, указав на ошибки, с подробным описанием проблемы и возможным способами решения. Иными словами, ошибки, которые в программах, написанные на других языках, возникли бы во время выполнения программы, в Rust даже не пройдут статическую проверку компилятора. Это позволяет программистам сосредоточится на логике программы, оставив «отлов» ошибок компилятору.

Отсутствие механизма сборки мусора, или наличие возможности её отключения. В Rust сборщика мусора нет вообще, это связано с тем, что он позиционируется как альтернатива системным языкам, таким как C/C++, и наличие механизма сборки мусора является недопустимым, так как это крайне негативно сказывается на производительность и общем размере исполняемых файлов.

Наличие программных механизмов, реализации идиомы RAII для управления памятью. Но без сборки мусора очень сложно писать код, так как приходится постоянно вручную управлять памятью, в связи с этим идиома

RAII заложена в основу Rust, а именно в её системе «владения», которая подразумевает что после присвоения какого-либо объекта переменной, этот объект рекурсивно связывается с этой переменной, и после того переменная покидает область видимости программы, то объект автоматически рекурсивно удаляется. То есть программисту вовсе не нужно думать об управлении памятью, управление заложено в основу языка, и не имеет никакого отрицательного влияния на производительность.

Поддержка многопоточного программирования. Rust всегда стремился быть модульным языком. Это значит, что по умолчанию, в нём доступны лишь самые базовые функциональные возможности, которые можно ожидать от языка. Весь остальной функционал доступен в качестве библиотек, этот факт делает Rust очень гибким и адаптивным к самым различным задачам. Программист сам решает какие функции ему нужны, а какие нет. Это же касается и поддержки многопоточного программирования, на сегодняшний день существует множество различных библиотек, с разными уровнями абстракций, каждая из которых оптимизирована на максимально эффективное использование вычислительных ресурсов эксплуатируемой системы.

Поддержка асинхронного программирования. В конце 2019-го года в Rust стал доступен синтаксис для написания асинхронного кода, подобно тому что доступен в других языках, таких как JavaScript или Python. Для асинхронного ввода/вывода существует перечень достаточно зрелых библиотек. Все гарантии безопасности, которые Rust даёт при написании синхронного кода, распространяются и на асинхронный код, таким образом, например, в Rust невозможно написать код, который будет иметь состояние гонки.

Из всего вышеописанного, можно сделать заключение, что язык программирования Rust подходит в качестве основного языка для разработки

системы веб-шаблонов, ввиду того что он удовлетворяет всем поставленным требованиям.

2.7. RESTful сервис, выполняющий роль системы веб шаблонов

В предыдущем разделе, в качестве основного языка разработки был выбран Rust, что приводит к возникновению ещё одной проблемы – проблемы интероперабельности с существующей ИТ-инфраструктурой предприятий. Как показывает практика, средние и крупные компании имеют в составе своей ИТ-инфраструктуры программные решения, написанные на нескольких языках программирования, как правило это высокоуровневые языки, такие как Java, Python, Kotlin, также встречаются и низкоуровневые такие как C++. Вообще говоря, это может быть любой язык программирования, и задача состоит в нахождении способа сделать разрабатываемую систему веб-шаблонов совместимой с любой IT-инфраструктурой, независимо от применяемых технологий.

TCP/IP и работающий поверх них HTTP протокол на сегодняшний день являются вездесущими, их поддержка реализована тем или иным способом и доступна во всех современных языках программирования. Это делает связку протоколов TCP/IP и HTTP отличным стандартизованным каналом для обмена информацией между сервисами, созданных с применением разных технологий. Существует даже целый архитектурный стиль написания подобного рода приложений, называемый REST, в основе которого лежит принцип использования стандарта HTTP для двустороннего взаимодействия различных компонентов распределенных систем. По своей сути REST это набор правил и ограничений, который определяет де-факто стандарт взаимодействия между веб-приложениями в вычислительной сети.

Таким образом, разрабатываемою систему веб-шаблонов можно спроектировать как отдельный веб-сервис, соблюдающий правила и не

нарушающий ограничений, накладываемых REST. Для взаимодействия с системой веб-шаблонов, будет необходимо разработать интерфейс программирования приложений – API, который в свою очередь будет содержать различные методы для предоставления услуг шаблонизации, иначе говоря, веб-сервис будет иметь клиент-серверную архитектуру. Так как API будет разрабатываться с учётом правил и ограничений REST, то разрабатываемый веб-сервис может называться RESTful.

Недостатком использования сети в качестве канала является относительно медленная передача данных между клиентом и сервером. Для нивелирования этой проблемы, можно располагать узлы в одной локальной сети, где скорость передачи данных, как правило значительно выше, а задержки передачи ниже. Ещё одним способом уменьшения времени передачи между клиентом и сервером является расположение их на одной вычислительной машине, что позволит им производить обмен информацией миную сетевые интерфейсы.

ГЛАВА 3. ПРОЕКТИРОВАНИЕ СЕРВИСА ПО ПРЕДОСТАВЛЕНИЮ УСЛУГ СИСТЕМЫ ВЕБ-ШАБЛОНОВ

3.1. Требования, предъявляемые к качеству разрабатываемой системы

Перед тем как начать проектировать систему веб-шаблонов, необходимо определить перечень требований к качеству, которым должная удовлетворять целевая система. В «ГОСТ Р ИСО/МЭК 25010 Модели качества систем и программных продуктов» представлены две модели качества: модель качества при использовании и модель качества продукта. Целесообразным решением будет использовать именно эти модели для определения требований к разрабатываемой системе.

Модель качества при использовании содержит пять основных характеристик качества, структура модели изображена на рис. 3.1.1.

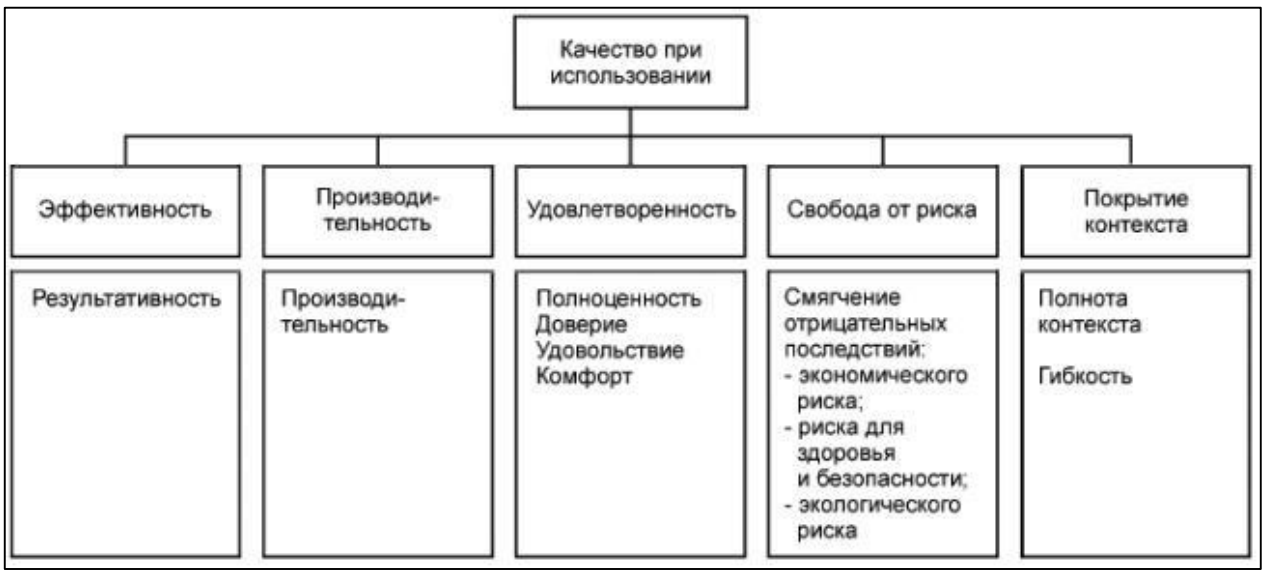

Рис. 3.1.1 Модель качества при использовании

Из приведённых характеристик качества, к разрабатываемой системе можно предъявить требования только по трём из них:

1. Эффективность, результативность. Точность и полнота, с которой пользователи достигают определенных целей.

2. Производительность. Связь точности и полноты достижения пользователями целей с израсходованными ресурсами.

3. Удовлетворенность. Способность продукта или системы удовлетворить требованиям пользователя в заданном контексте использования.

Модель качества продукта содержит восемь основных характеристик качества, иерархическая модель данной модели изображена на Рис. 3.1.2

Рис. 3.1.2 Модель качества продукта

Из приведённых характеристик качества продукта, к разрабатываемой системе можно предъявить требования по пяти из них:

1. Уровень производительности. Производительность относительно суммы использованных при определенных условиях ресурсов.
2. Совместимость. Способность продукта, системы или компонента обмениваться информацией с другими продуктами, системами или компонентами, и/или выполнять требуемые функции при совместном использовании одних и тех же аппаратных средств или программной среды.
3. Удобство использования. Степень, в которой продукт или система могут быть использованы определенными пользователями для

достижения конкретных целей с эффективностью, результативностью и удовлетворенностью в заданном контексте использования.

4. Надежность. Степень выполнения системой, продуктом или компонентом определенных функций при указанных условиях в течение установленного периода времени.
5. Сопровождаемость, модифицируемость. Результативность и эффективность, с которыми продукт или система могут быть модифицированы предполагаемыми специалистами по обслуживанию.

Все описанные требования в дальнейшем будут оказывать влияние на выбор определённых технологий для разработки проекта, и на архитектурные решения во время его проектирования

3.2. Моделирование работы сервиса

Переда началом разработки необходимо спроектировать архитектуру веб-сервиса, и смоделировать его работу. Как говорилось ранее, веб-сервис будет иметь клиент-серверную архитектуру, то есть приложения клиенты будут отправлять HTTP запросы на сервер, сервер будет обрабатывать запрос и посылать обратно HTTP ответ. Такая архитектура позволяет удовлетворить требование совместимости с уже существующей ИТ инфраструктурой. Ещё одним преимуществом такого подхода является удобство использования, так как от приложения клиента требуется лишь поддержка HTTP протоколов обмена данными. Для повышения характеристики сопровождаемости веб-сервис необходимо разрабатывать модульным, то есть за каждый конкретный тип запроса будет отвечать отдельный модуль, и для расширения возможностей веб-сервиса необходимо будет просто разработать новый модуль с минимальными изменениями в остальных частях приложения.

Так как взаимодействовать исключительно через API может быть неудобным для специалистов не технического профиля и даже прикладным

программистам, то целесообразно будет разработать веб-интерфейс, который позволит интерактивно взаимодействовать с веб-сервисом, например, для проведения отладки шаблона или создания совершенно новых шаблонов.

Использование многопоточного и асинхронного программирования в разработке сервиса позволит ему обрабатывать огромное количество запросов одновременно даже на машинах с ограниченными вычислительными ресурсами.

Далее будет проведено моделирование работы разрабатываемого веб-сервиса с использованием функциональных диаграмм в нотации IDEF0. Для создания диаграмм использовалась программа Ramus Educational версии 1.1.

На самом верхнем уровне, работа веб-сервиса выглядит следующим образом (Рис. 3.2.1).

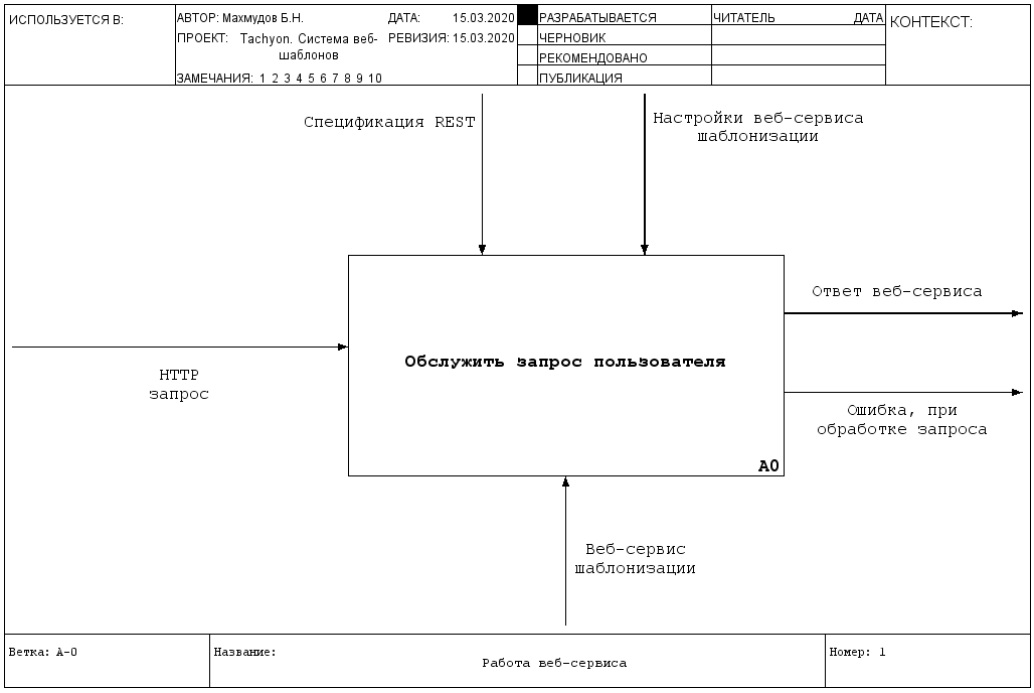

Рис. 3.2.1 Контекстная диаграмма работы веб-сервиса

На вход поступает HTTP запрос от клиента, веб-сервис шаблонизации обрабатывает его, руководствуясь спецификацией REST и настройками

самого сервиса, и на выходе он выдаёт ответ, который содержит запрашиваемый ресурс или ошибку, если запрос не может быть удовлетворён.

Декомпозирую этот уровень более детально, можно выделить основные блоки (модули) которые принимают участие в обработке запроса (Рис. 3.2.2).

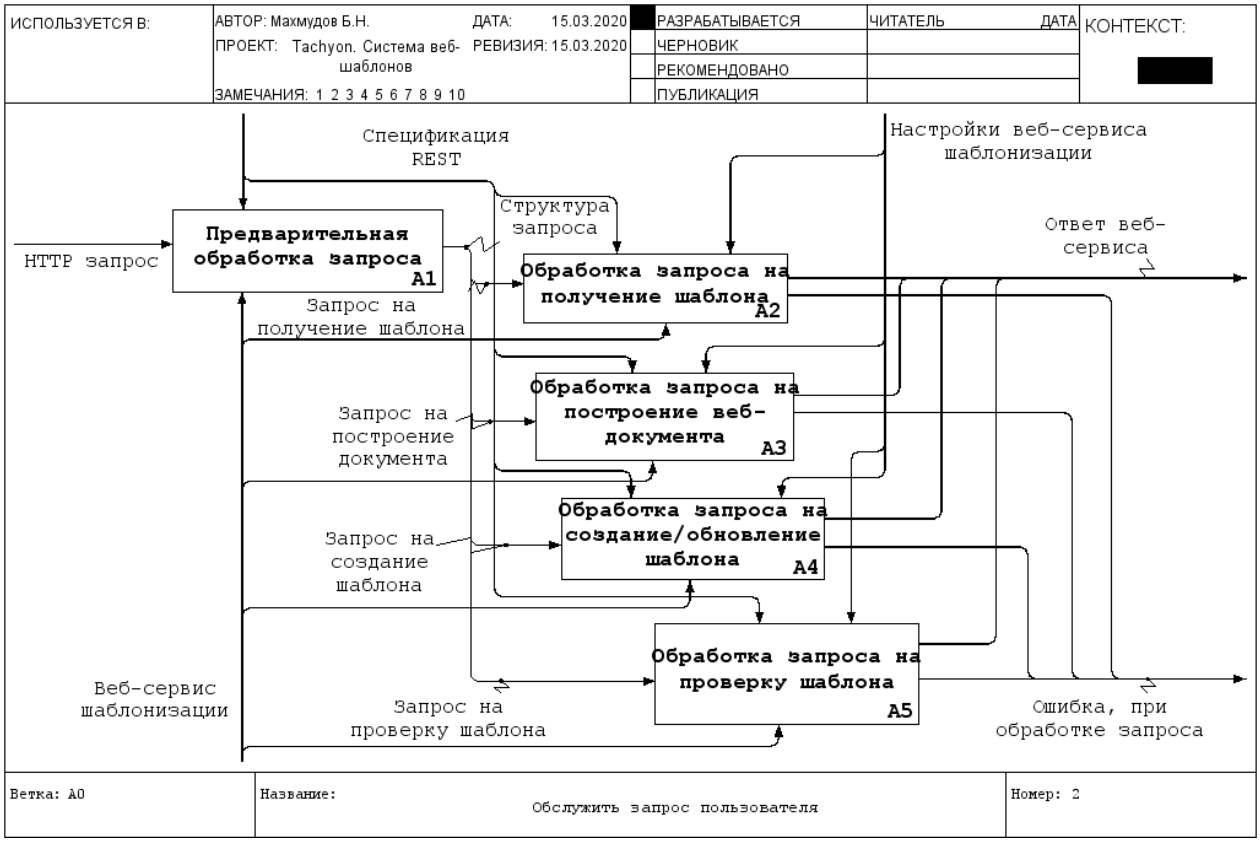

Рис. 3.2.2 Декомпозиция контекстной диаграммы

Как видно из диаграммы, запрос клиента сначала подвергается предварительной обработке для конвертации его во внутреннее представление запроса, с которым могут работать другие модули. Далее в зависимости от типа запроса он отдаётся на обработку соответствующему модулю. На диаграмме изображено четыре модуля обработки, но вообще говоря их может быть больше. Каждый модуль может быть декомпозирован на более детализированный уровень, что и будет сделано далее.

На Рис. 3.2.3 приведена DFD декомпозиция блока предварительной обработки запроса. Так как HTTP запрос представляет из себя поток байт, то

его сначала необходимо считать с сокета установленного подключения, после чего это множество байт разбирается в составляющие компоненты запроса, и из этих компоненты собирается внутреннее представление для дальнейшей обработки.

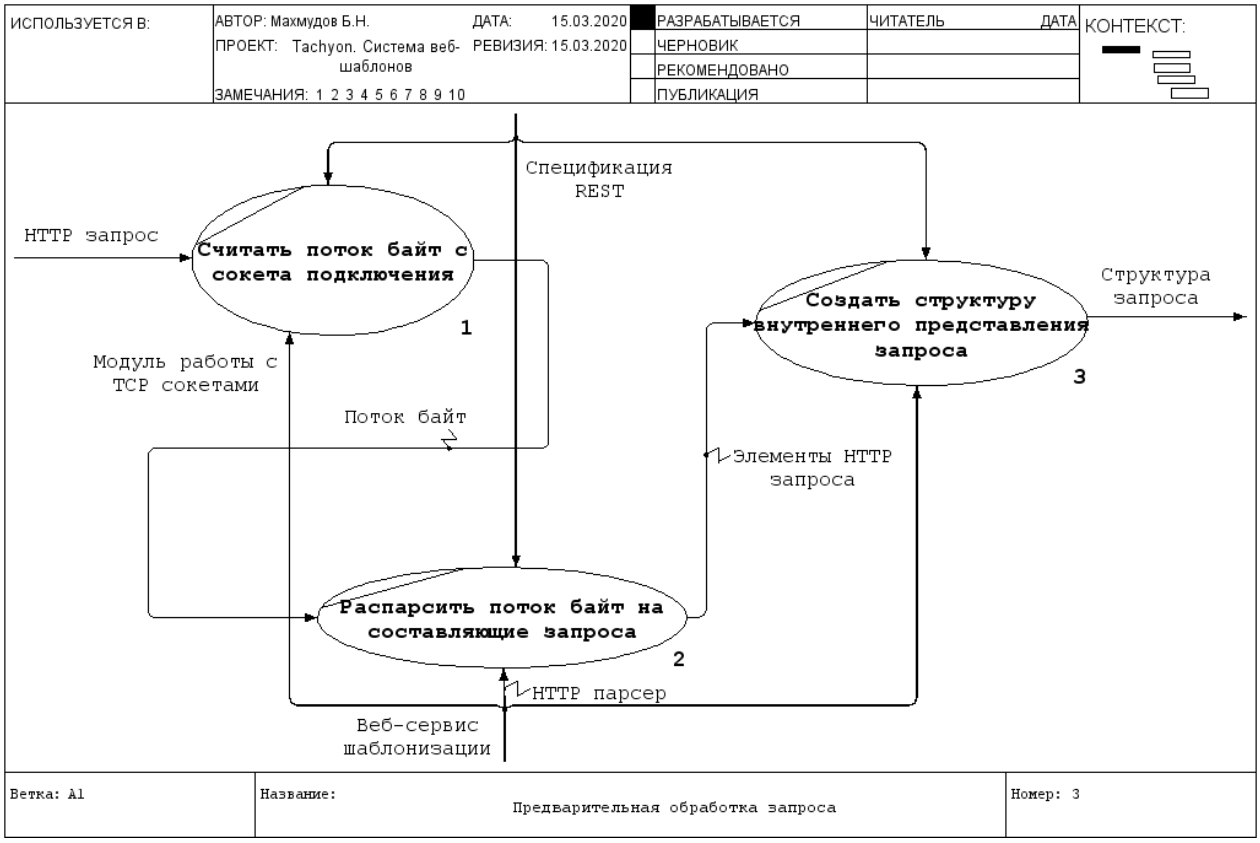

Рис. 3.2.3 DFD декомпозиция предварительного обработки запроса

Из полученной структуры данных можно определить то, как нужно обработать запрос, и эта структура отдается соответствующему модулю. На Рис. 3.2.4 продемонстрирована диаграмма, которая моделирует процесс обработки запроса на получение шаблона. Для этого необходимо получить уникальный идентификатор шаблона, проверить его наличие в специальном файле индексе, который содержит всю информацию о шаблоне. В случае нахождения шаблона в индексе, его надо считать из файла шаблона и сформировать ответ в соответствии со спецификацией REST, в противном случае клиенту отправляется ошибка об отсутствии шаблона.

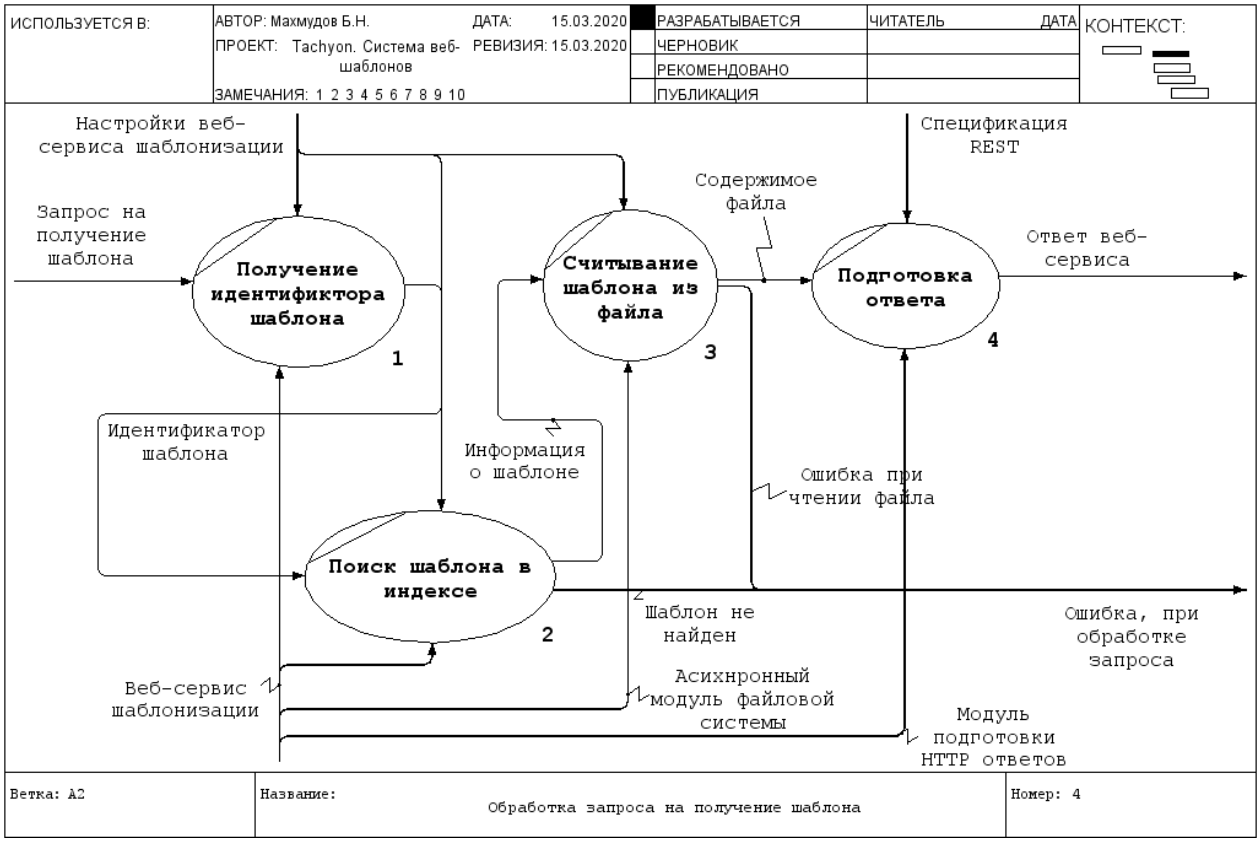

Рис. 3.2.4 DFD декомпозиция процесса получения шаблона

На Рис. 3.2.5 изображена IDEF0 декомпозиция блока по построению веб-документа (основная функция проектируемого веб-сервиса). Шаблонизатору для построения конечного веб-документа необходимо наличие абстрактного синтаксического дерева и данных шаблонизации. Так как для ускорения процесса шаблонизации была введена особая структура данных для хранения уже построенных АСД – кеш, то перед тем как загружать шаблон из файла необходимо сделать проверку наличия АСД в кеше. Если шаблон отсутствует в кеше, нужно выполнять все шаги по построению АСД сначала, в противном случае можно сразу вернуть готовое АСД, данный процесс изображён на Рис. 3.2.6. После получения абстрактного синтаксического дерева, необходимо сделать валидацию данных в соответствии с настройками шаблонизатора. Последним шагом

является непосредственное построение конечного веб-документа и формирование ответа сервиса (Рис. 3.2.7).

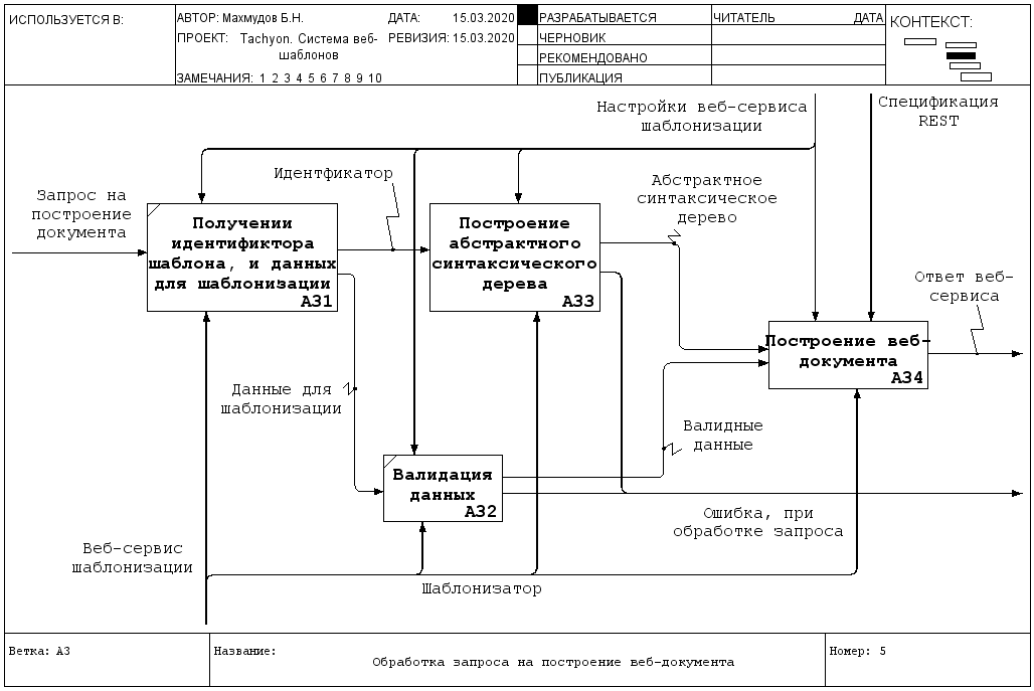

Рис. 3.2.5 IDEF0 декомпозиция процесса генерации веб-документа

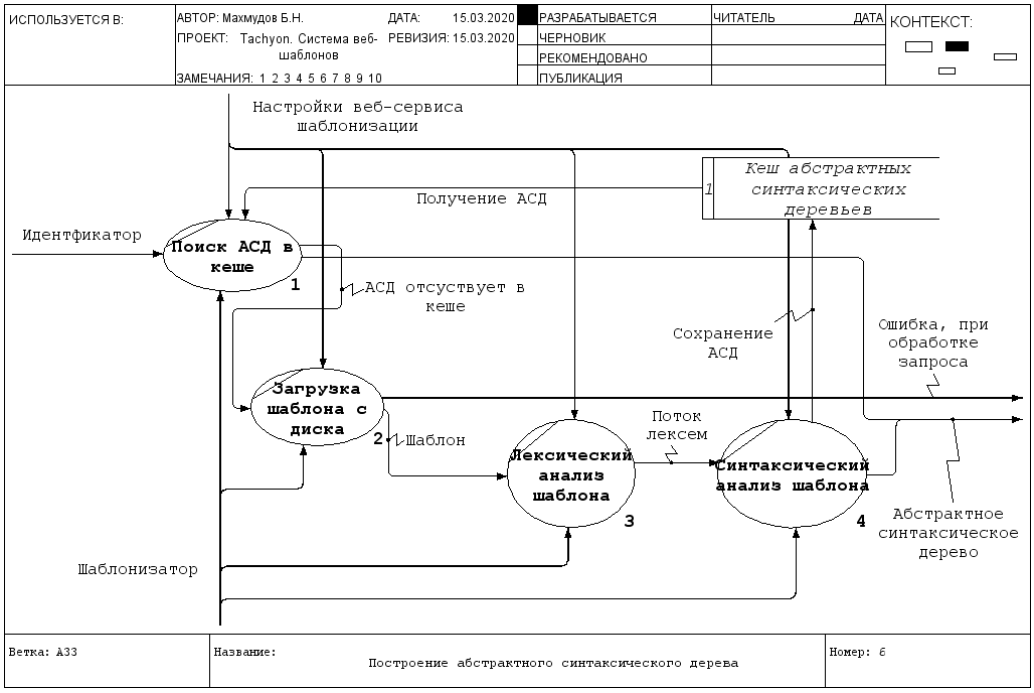

Рис. 3.2.6 DFD декомпозиция процесса получения абстрактного синтаксического дерева

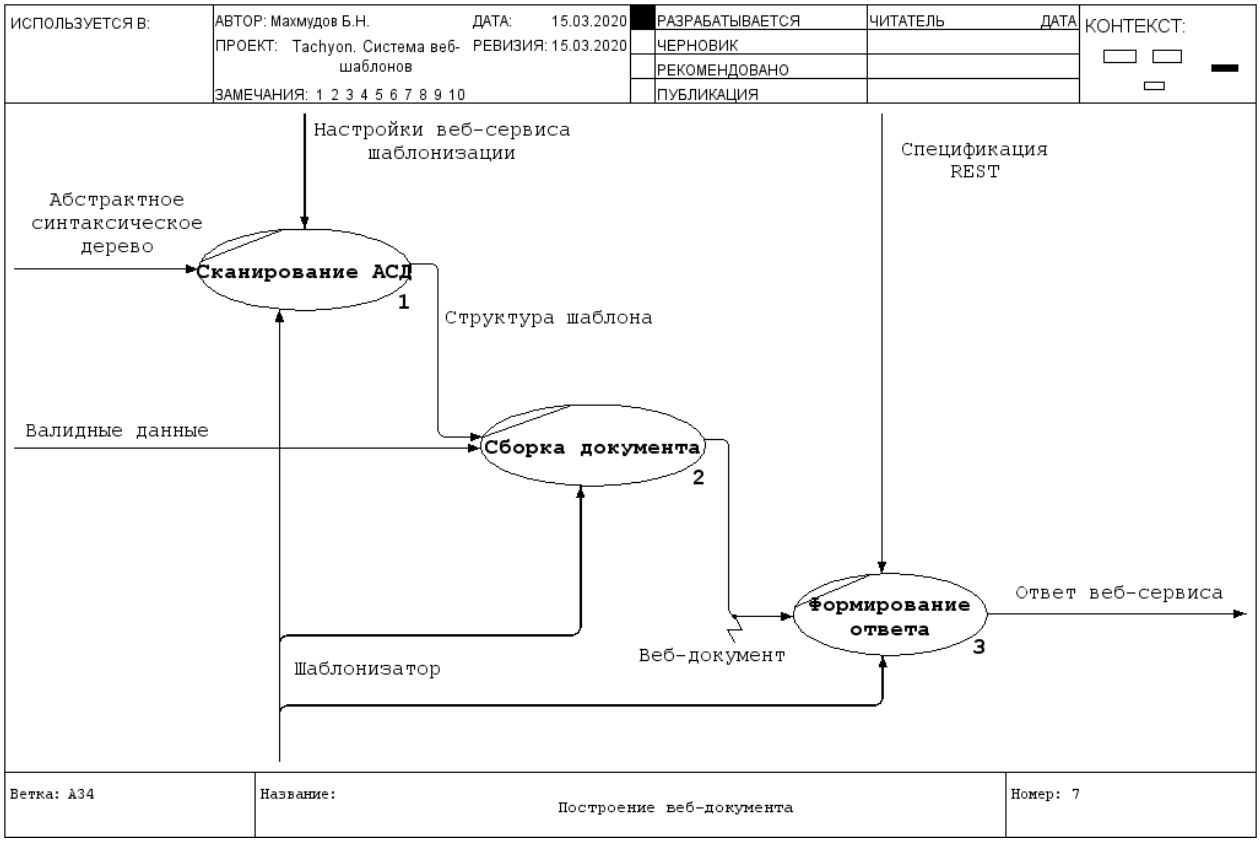

Рис. 3.2.7 DFD декомпозиция процесса построения документа

Рис. 3.2.8 демонстрирует относительную сложную диаграмму, моделирующую процесс создания или обновления шаблона. Для начала необходимо сделать проверку самого шаблона на корректность путём анализа его шаблонизатором. Если шаблон корректен, то далее нужно найти его в индекс файле, и если он там присутствует, то нужно обновить файл шаблона новыми данными, если же шаблон отсутствует в индекс файле, значит необходимо создать новый шаблон, создав новый файл шаблона. В конце необходимо сохранить изменения в индекс файле и уведомить клиента об успешном сохранении. Если в каком-то из шагов произойдёт ошибка, то никакие изменения не будут сохранены в постоянную память, тем самым не нарушая целостность системы, а клиенту будет отправлено соответствующее сообщение о возникшей ошибке обработки запроса.

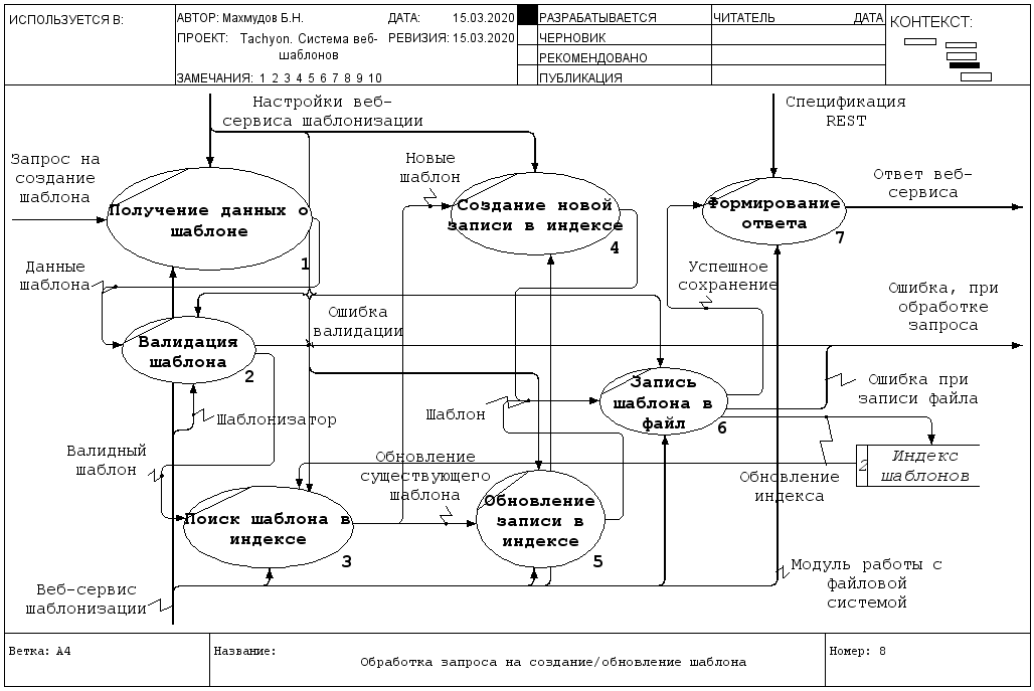

Рис. 3.2.8 DFD декомпозиция процесса создания или обновления шаблона

На Рис. 3.2.9 приведена диаграмма процесса проверки шаблона на корректность в нотации DFD. Процесс аналогичен тому, что был описан при создании или обновлении шаблона.

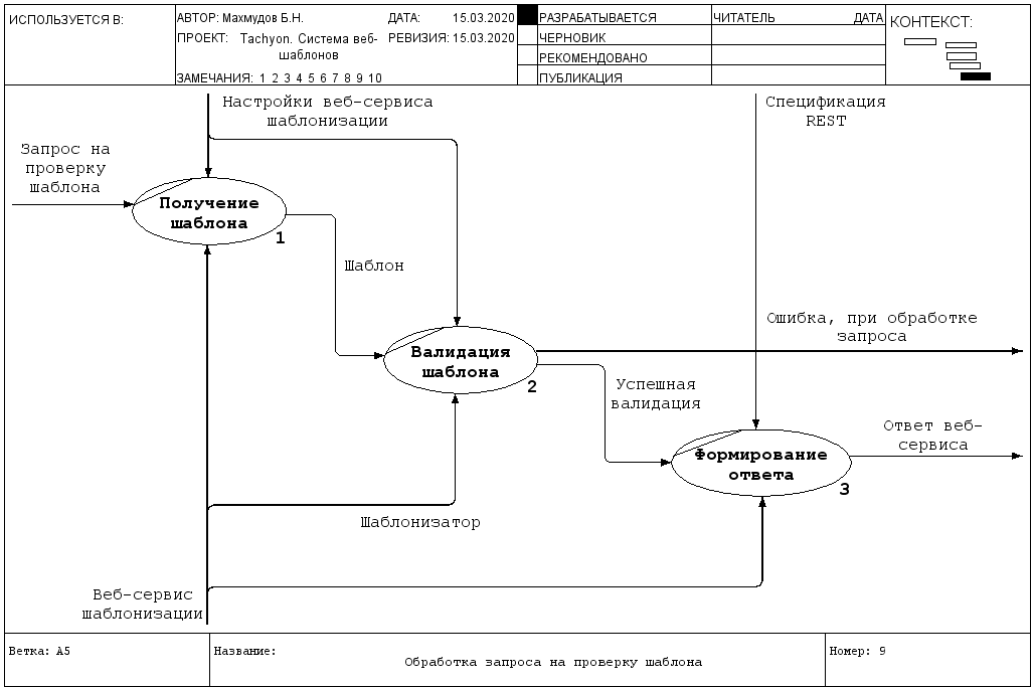

Рис. 3.2.9 DFD декомпозиция проверки веб-шаблона

3.3. Выбор инструментов

Перед началом разработки следует уделить тщательное внимание выбору инструментов и технологий, которые будут использованы для реализации проекта. В силу того, что разрабатываемое приложение имеет клиент-серверную архитектуру, где клиент может быть любой сущностью, поддерживающая HTTP протокол для обмена данными, то разработка клиента не относиться к целям данной выпускной квалификационной работы, выбор клиента остаётся на усмотрение конечного пользователя. Но как оговаривалось ранее, для удобства использования сервиса, требуется разработать веб-интерфейс для интерактивного взаимодействия с системой шаблонизации. С учётом сказанного разработку можно разделить на два области: разработка серверной части или бэкенд составляющей и разработка графического пользовательского интерфейса или фронтенд составляющей.

Для разработки серверной части будет использован язык программирования Rust с библиотеками из экосистемы языка, а именно необходимы будут следующие библиотеки:

- библиотека обработки HTTP запросов,
- библиотека, предоставляющая многопоточную и асинхронную среду выполнения для разрабатываемого приложения,
- библиотека сериализации и десериализации структур данных Rust в форматы обмена данными, таких как JSON или XML,
- библиотека шаблонизации.

В качестве обработчика HTTP запросов была выбрана библиотека httparse, в силу того, что библиотека почти полностью избегает выделений памяти и использует низкоуровневые SIMD инструкции процессора для ускорения процесса разбора запроса.

Для предоставления многопоточной и асинхронной среды выполнения будет использована достаточно зрелая библиотека tokio, так как она предоставляет множество абстракций над низкоуровневыми механизмами управления потоками исполнения и ввода/вывода данных, при этом не жертвуя производительностью.

Для сериализации и десериализации структур данных Rust будет использована библиотека serde, ввиду наличия хорошей документации, и ориентации на высокие показатели быстродействие.

Основным компонентом системы веб-шаблонов является шаблонизатора, из-за чего необходимо выбрать одновременно быструю и удобную в использовании библиотеку шаблонизации. В экосистеме Rust существует два вида шаблонизаторов, те, которые компилируют шаблоны вместе с исполняемым файлом, и те, которые анализируют шаблон во время выполнения программы. Причем в первом случае процесс построения веб-документов на порядок быстрее чем во втором, но такой подход имеет два существенных изъяна:

1. Время компиляции программы, которое и без того достаточно большое, становится ещё больше в зависимости от количества шаблонов.
2. Отсутствует возможность динамического добавления или обновления шаблонов.

Так как оба этих недостатка недопустимы в случае разрабатываемой системы, то выбирать придётся из второй группы. Среди таких библиотек, согласно нескольким тестам на производительность, лидирующую позицию занимает библиотека Tera, которая обладает синтаксисом языка шаблонизации схожую с Django. Также Tera имеет очень подробную документацию по использованию, как самой библиотеки, так и языка шаблонизации.

Для разработки графического пользовательского интерфейса будут использована совокупность технологий веб-разработки HTML/CSS/JavaScript. Для облегчения разработки также будут применены дополнительные фреймворки, которые основаны на этих технологиях, а именно:

- Element-UI, HTML/CSS фреймворк для создания веб-интерфейсов,
- Vue.js JavaScript фреймворк для создания динамичных и асинхронных веб-интерфейсов.

В следующем разделе будет описан процесс разработки веб-сервиса, с использование вышеописанных технологий.

3.4. Разработка сервиса

По причине того, что архитектурно само приложение можно разделить на бэкенд и фронтенд составляющие, то и саму разработку удобно проводить в два отдельных этапа:

1. Разработка серверной части (бэкенда).
2. Разработка графического интерфейса пользователя (фронтенда).

Работу над созданием серверной части можно подразбить на отдельные подзадачи:

- разработке модуля шаблонизации, будет являться ядром сервиса,
- разработке общего интерфейса обработки ошибок, с учетом спецификации REST.
- разработке модуля работы с файловой системой, а именно реализация базовых возможностей веб-сервера,
- разработке асинхронной многопоточной среды для работы сервиса.

Модуль шаблонизации полностью базируется на библиотеке Tera. Задачей модуля является обработка запросов на построение документа с

использование указанного в запросе шаблона, и предоставленных данных. Основой модуля является структура данных представляющая из себя экземпляр шаблонизатора и все ассоциированные с ним данные. Эта структура данных является глобальной и статичной, и может быть безопасно использована множеством обработчиков запросов одновременно, ввиду того, что она не изменяема и гарантированно будет сохранять одно и тоже состояние. Для безопасного изменения этой структуры данных, например, при добавлении или изменении шаблонов, необходимо будет накладывать на структуру какую-либо блокировку с целью недопущения порчи данных. Для таких целей в стандартной библиотеке Rust имеется особая разновидность блокировок, называемая RWLock, её отличие от мютексов состоит в том, что мютексом одновременно может владеть только один поток исполнения, вне зависимости от того, используется ли он для чтения или записи, а RWLock, может одновременно иметь множество потоков, которые обращаются к содержимому для чтения или только один поток, который может обращаться к содержимому для изменения. Таким образом в стандартном режиме множество потоков могут одновременно использовать структуру данных шаблонизатора, которая обёрнута в RWLock, для построения документов, но для изменения шаблонов, изменяющему потоку нужно будет дождаться другие читающие потоки, до завершения их работы, и только после этого получить эксклюзивную блокировку и произвести изменение. Похожее поведение имеется в системах управления базами данных без поддержки много-версионности.

Стандартизованная обработка ошибок также является важной задачей, в силу того, что ошибки в ходе работы сервиса могут возникать в самых разных ситуациях, далее приведен пример наиболее возможных ошибок:

- ошибка чтения из файла или записи в файл;

- ошибка получения запроса от клиента;
- ошибка валидации данных для шаблонизации;
- ошибка проверки шаблона на корректность.

Все эти ошибки возникают в разных программных модулях и вообще говоря имеют разную структуру. Основная задача состоит в том, чтобы привести все эти ошибки к единому формату для дальнейшего формирования ответа об ошибке для клиента. В Rust не имеется механизмов отлова ошибок, которые присутствуют в других языках, таких как try-catch в JavaScript или try-except в Python, сделано это всё опять-таки для повышения производительности. Вместо этого в Rust принято возвращать результат операции, который представляет из себя алгебраический тип данных, то есть составной тип, который может быть либо ожидаемым значением, которая должна произвести операция, если она завершилась успехом, или же ошибкой, если операция завершилась неудачно. Также, как и в других языках в Rust имеется механизм распространения ошибок вверх по стеку вызовов. То есть если ошибка произошла в какой-либо функции, то программист может решить передать ошибку вызывающей функции, а та в свою очередь вызвавшей её, и так далее до обработчика ошибок, который извлечёт необходимую информацию и сформирует ответ, уведомляющий пользователя о произошедшей ошибке.

Возможность работать с файловой системой необходима для:

1. возможность считывать и записывать файлы шаблонов;
2. возможность обслуживать запросы на получение статических файлов пользовательского интерфейса.

Второй случай использование позволяет разрабатываемому приложение выступать в качестве базового веб-сервера для обслуживания запросов на статичные файлы, например, это могут быть HTML страницы, файл таблицы

стилей или файл, содержащий JavaScript код. При этом нужно учесть, что скорость чтения и записи файлов может быть значительно ниже чем возможности сервиса обработке запросов. В связи с этим, дабы не допустить простоя вычислительных ресурсов и появления очередей запросов, необходимо разработать асинхронную неблокирующую среду выполнения программы.

Для создания асинхронной среды выполнения используется библиотека tokio, которая построена вокруг библиотеки futures, предоставляющая элементарные единицы асинхронных вычислений. Future – это значение какого-либо вычисления, которое ещё не завершилось, аналогично Promise в JavaScript, но с оптимизациями, направленные на более эффективное использование ресурсов. Основной механизм работы Future можно описать следующим образом:

1. Программист описывает структуру данных, и ассоциированные с ней методы, которая представляет из себя, строго говоря, конечный автомат, который может переходить из одного состояния в другое под влиянием каких-либо событий, происходящих в системе. Состояния соответствуют этапам вычисления, через которые оно должно пройти для получения конечного результата.

2. Описание этого вычисления отдаётся на исполнение среде выполнения, называемой executor. Executor будет пытаться выполнить все этапы вычисления поочерёдно, но может столкнуться с блокировками, например связанными с вводом/выводом. В таком случае, среда выполнения сохранит состояние конечного автомата, с регистрацией запроса операционной системе, на уведомлении о наступлении требуемого события, например, завершения операции ввода/вывода. После этого среда выполнения возьмётся за исполнение другого вычисления, которое может быть

продолжено. С наступлением этого события executor продолжит выполнение сохранённого ранее вычисления с того этапа, с которого оно было прервано. Executor одновременно можно использовать несколько потоков выполнения, тем самым максимально эффективно используя вычислительные ресурсы.

Веб-сервис будет компилироваться в один исполняемый файл, который не имеет сторонних зависимостей, так как все необходимые компоненты уже включены в состав бинарного файла. Этот факт делает приложение очень простым в использовании. От пользователя требуется лишь указать на каком сетевом порту нужно запуститься веб-сервису, что делается из командной строки.

Следующим этапом разработки является создание графического пользовательского интерфейса. Интерфейс пользователя будет предоставлять базовые возможности, такие как:

- построение документов на основе существующих шаблонов, и передаваемых пользователем данными;
- ознакомление со списком существующих шаблонов;
- редактирование шаблонов;
- создание новых шаблонов;
- ознакомление с документацией по синтаксису языка шаблонизации;
- ознакомление с документацией по методам, предоставляемых API веб-сервиса.

Основная часть разработки интерфейса будет производится с использованием JavaScript фреймворка Vue.js. Данный фреймворк за последние годы, хорошо зарекомендовал себя как средство создания SPA (Single Page Application), то есть веб-приложений в одной странице. Иначе говоря, при посещении страницы, все необходимые компоненты веб-приложения грузятся с сервера с самого начала, и дальнейшее

взаимодействие с сервером будет производиться асинхронно без перезагрузки страницы. За каждую функциональную возможность в Vue.js отвечает отдельный компонент, и так как всего графический интерфейс предоставляет шесть различных функциональных возможностей, то целесообразно для каждой разработать отдельный компонент. Каждый компонент включает в себя три составляющих: html верстка страницы, стили страницы, логика страницы, написанная на JavaScript с использованием Vue.js.

При обращении к сервису, пользователь попадает на главную страницу, которая изображена на Рис. 3.4.1. На этой странице имеется панель навигации, посредством которой можно выбирать интересующий пользователя инструмент.

Рис. 3.4.1 Страница по умолчанию, куда попадает пользователь

Далее можно воспользоваться возможностями генерации документа на основе существующих шаблонов. На Рис. 3.4.2 изображён интерфейс построение документа, основном элементом страницы является инструмент для ввода JSON, данный инструмент поддерживает несколько режимов отображения, таких как дерево, код, или форма, также в нем реализована

проверка введённого JSON на корректность. В правой части страницы находятся кнопки загрузки шаблона из файла, запуска генерации документа и перегрузки шаблонов. Помимо этого, в этой же части находится выпадающий список со списком доступных шаблонов. После ввода данных в формате JSON и выбора интересующего шаблона можно запустить генерацию веб-документа.

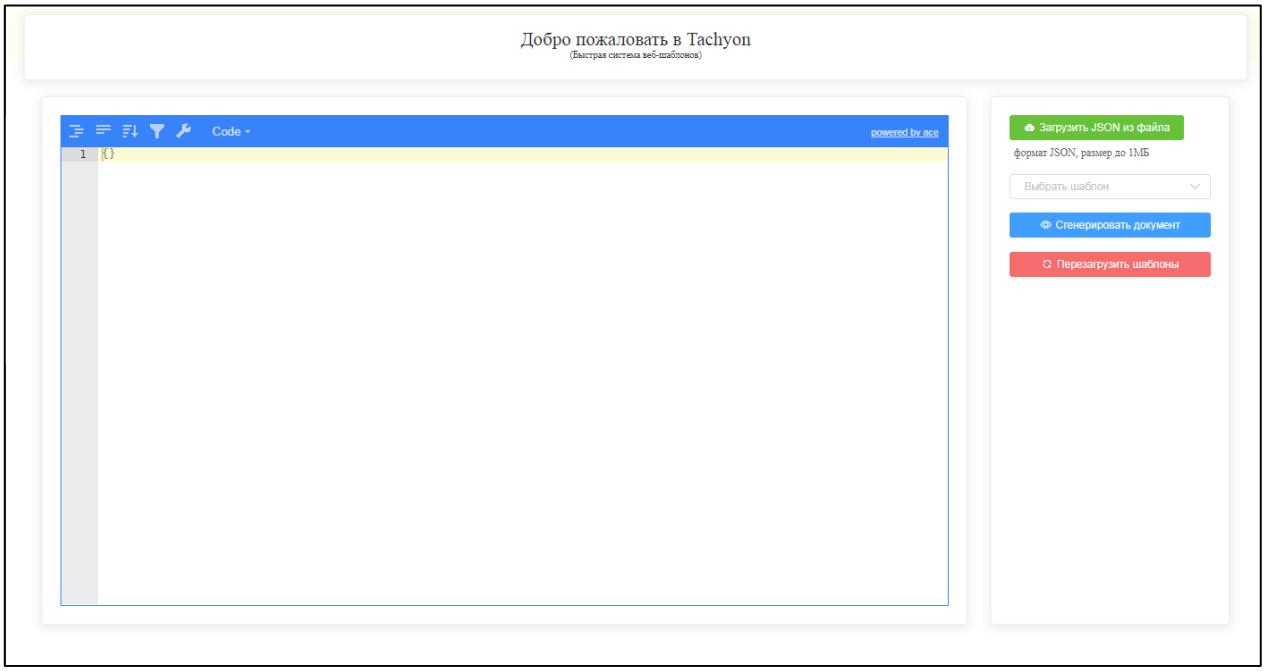

Рис. 3.4.2 Интерфейс ввода данных для построения документа

На Рис. 3.4.3 продемонстрирован сгенерированный документ, который выводится поверх интерфейса ввода данных.

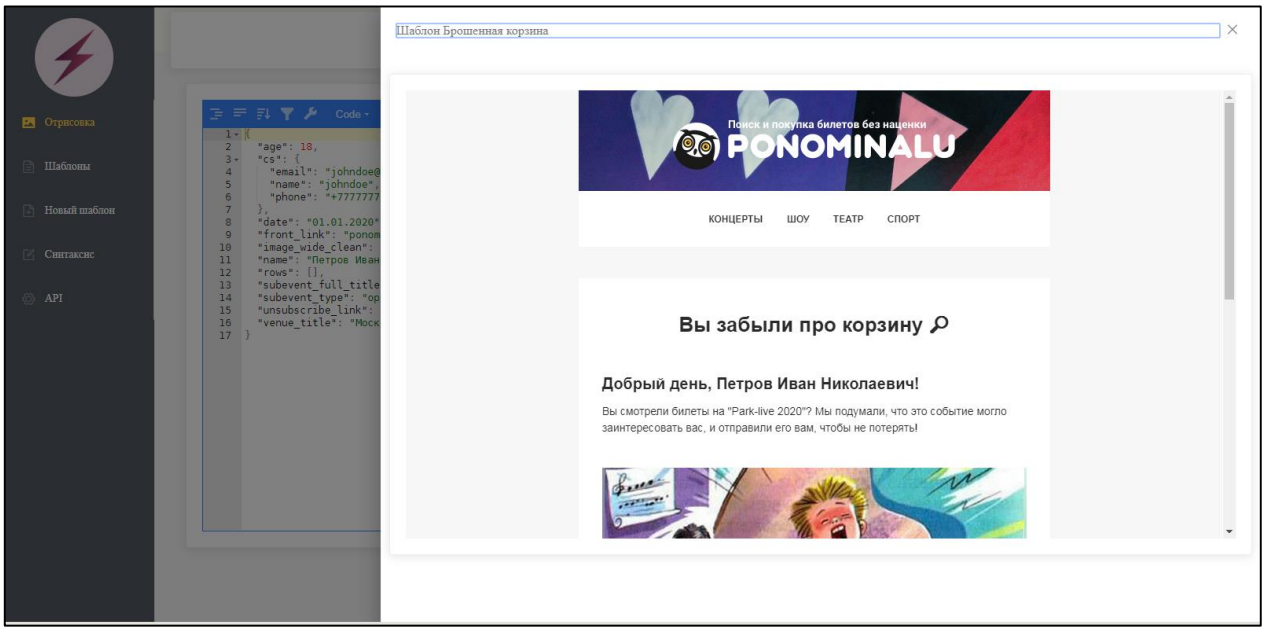

Рис. 3.4.3 Сгенерированный документ, поверх интерфейса ввода данных

Чтобы подробно ознакомится со списком доступных шаблонов необходимо перейти в раздел «Шаблоны». Тут же можно отредактировать интересующий шаблон (Рис. 3.4.4).

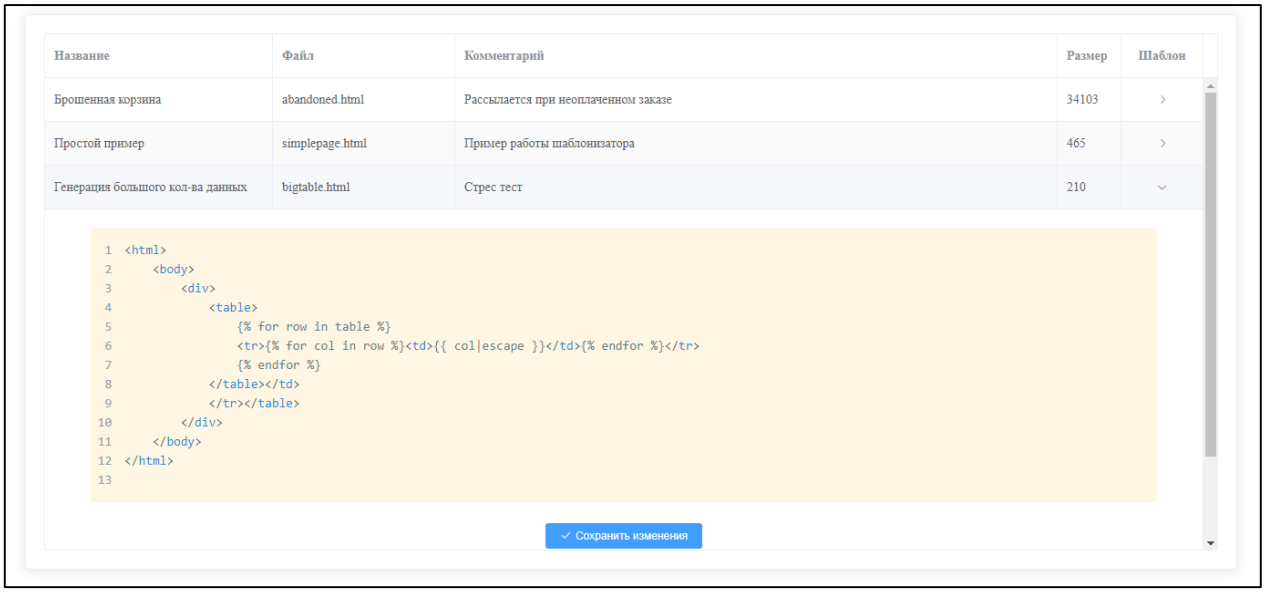

Рис. 3.4.4 Интерфейс со списком существующих шаблонов, и возможностью их редактирования

В случае если нужно добавить новый шаблон, то необходимо открыть раздел «Новый шаблон», интерфейс которого изображён на Рис. 3.4.5. Здесь доступен текстовый ввода шаблона, текстовый ввод тестовых данных, название шаблона, название файла шаблона и комментарий к шаблону. Также шаблон, как тестовые данные можно выгрузить из файла.

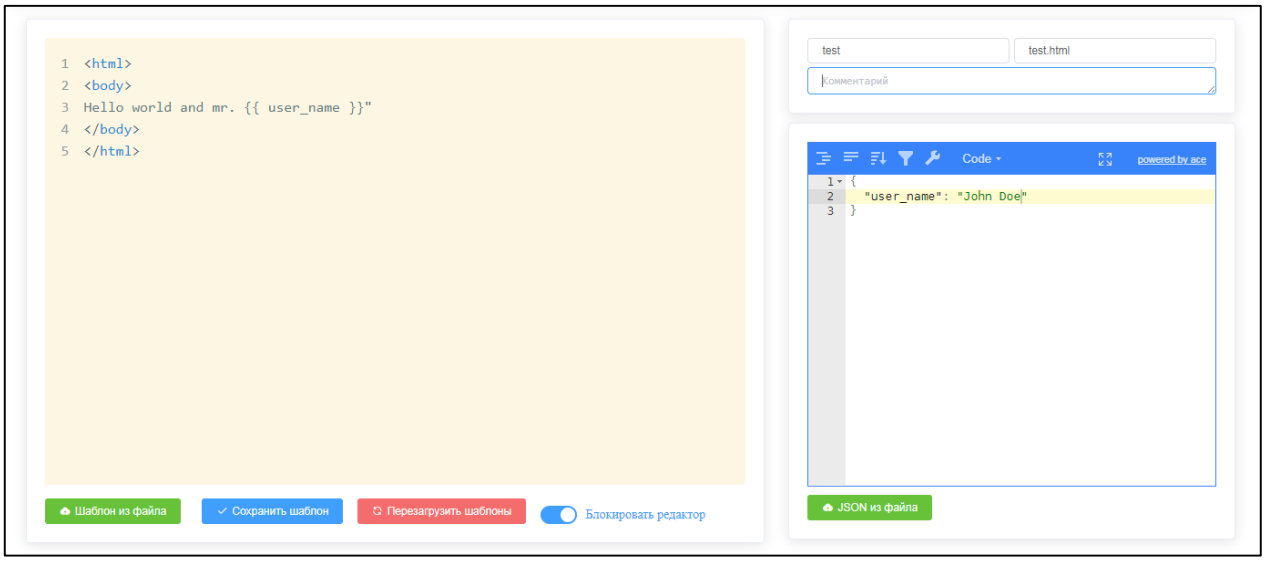

Рис. 3.4.5 Интерфейс создания нового шаблона

Для ознакомления с синтаксисом языка шаблонизации, требуется открыть раздел «Синтаксис» (Рис. 3.4.6), где каждая пункт подробно расписан с примерами. Похожая документация существует и для API, который предоставляет веб-сервис (Рис. 3.4.7).

Рис. 3.4.6 Интерфейс для ознакомления с документацией

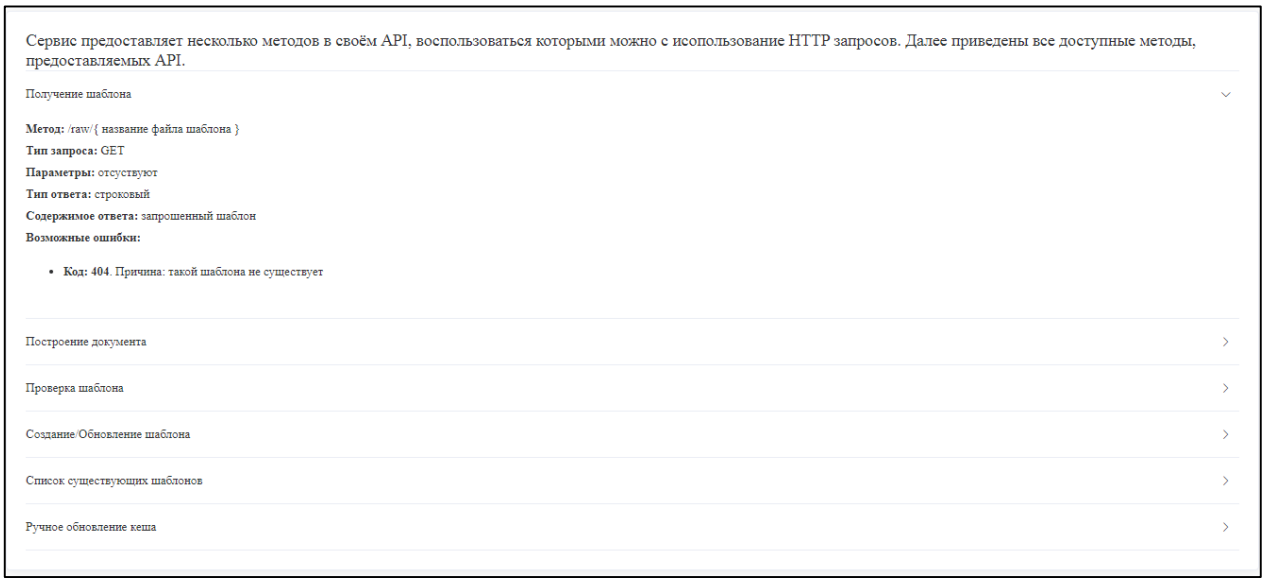

Рис. 3.4.7 Документация API

В результате был разработан асинхронный веб-сервис на языке Rust, который удобен в применении и теоретически должен обладать высокими показателями производительности. Для упрощения взаимодействия с веб-сервисом к нему был разработан веб-интерфейс, который позволяет использовать приложение людям не технической специальности. В следующем разделе будут произведены тесты, с целью оценки показателей производительности разработанной системы шаблонизации.

3.5. Результат разработки, оценка производительности

Для оценки производительности необходимо создать нагрузку на веб-сервер, которая будет имитировать реальную работу в условиях эксплуатации. Основная суть теста заключается в инкрементном увеличении количества одновременных запросов на API веб-сервиса, и параллельный мониторинг времени ответа веб-сервиса и количества потребляемых вычислительных ресурсов на сервере. Так как сложно создать интенсивный трафик из одной точки, для проведения теста будет использованы интернет сервисы по тестированию веб-приложений.

В целях проведения нагрузочных тестов, разработанное приложение было установлено на виртуальную машину в облачном сервисе Google Cloud Platform. Характеристики машины, следующие:

- двухъядерный процессор от Intel, поколения Kaby Lake, с тактовой частотой два гигагерца на ядро;
- два гигабайте оперативной памяти;
- гигабитный интернет канал;
- операционная система – Debian Stretch.

Для проведения первого теста использовался сервис K6 который позволяет без взимания оплаты проводить нагрузочные тесты с использованием до пятидесяти одновременных подключений. Настройки теста выглядят как изображено на **Ошибка! Источник ссылки не найден.**, нагрузка начнёт постепенно увеличиваться в течении пяти минут и в пиковый момент достигнет пятидесяти одновременных подключений. Обращение к серверу, который расположен в Стокгольме, будет производится из Франкфурта. Каждый запрос, сделанный к веб-сервису представляет из себя запрос на построение веб-документа, который отображает таблицу, содержащую две тысячи ячеек. Такой объём таблицы позволяет оценить производительность самого шаблонизатора. Результат теста изображён в виду графика на Рис. 3.5.2. По графику видно, что количество подключений постепенно возрастает, но при этом время ответа веб-сервиса остаётся постоянным в пределах семидесяти миллисекунд.

Рис. 3.5.1 Тест №1, настройки нагрузки

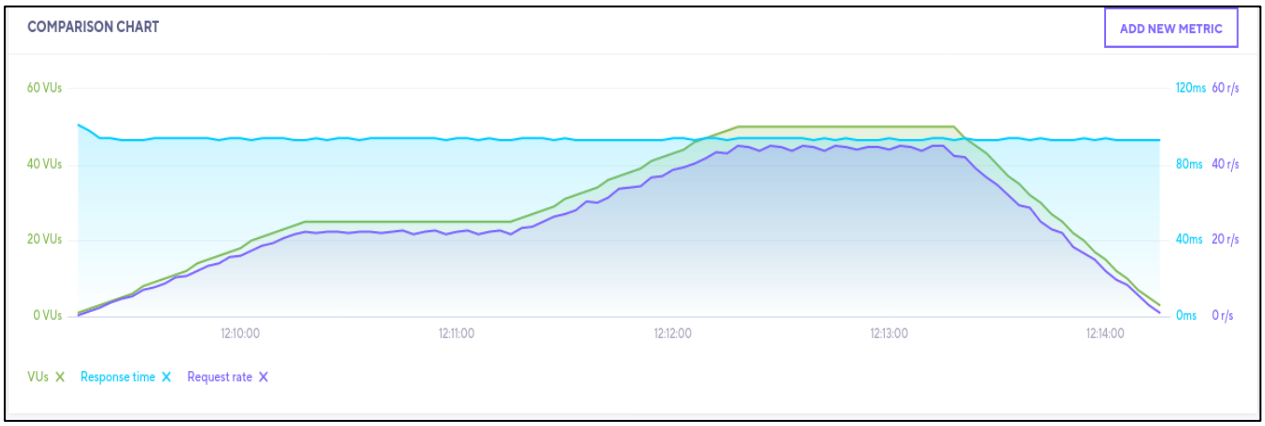

Рис. 3.5.2 Тест №1, 50 одновременных подключений

На Рис. 3.5.3 и Рис. 3.5.4 показано состояние вычислительной машины, на которой работает веб-сервис, по ним видно, что даже в момент максимальной нагрузки, потребление веб-сервисом процессорного времени не превышает 10%. Отсюда можно сделать вывод о том, что даже при пятидесяти одновременных подключениях, разработанный веб-сервис хорошо справляется со своей задачей.

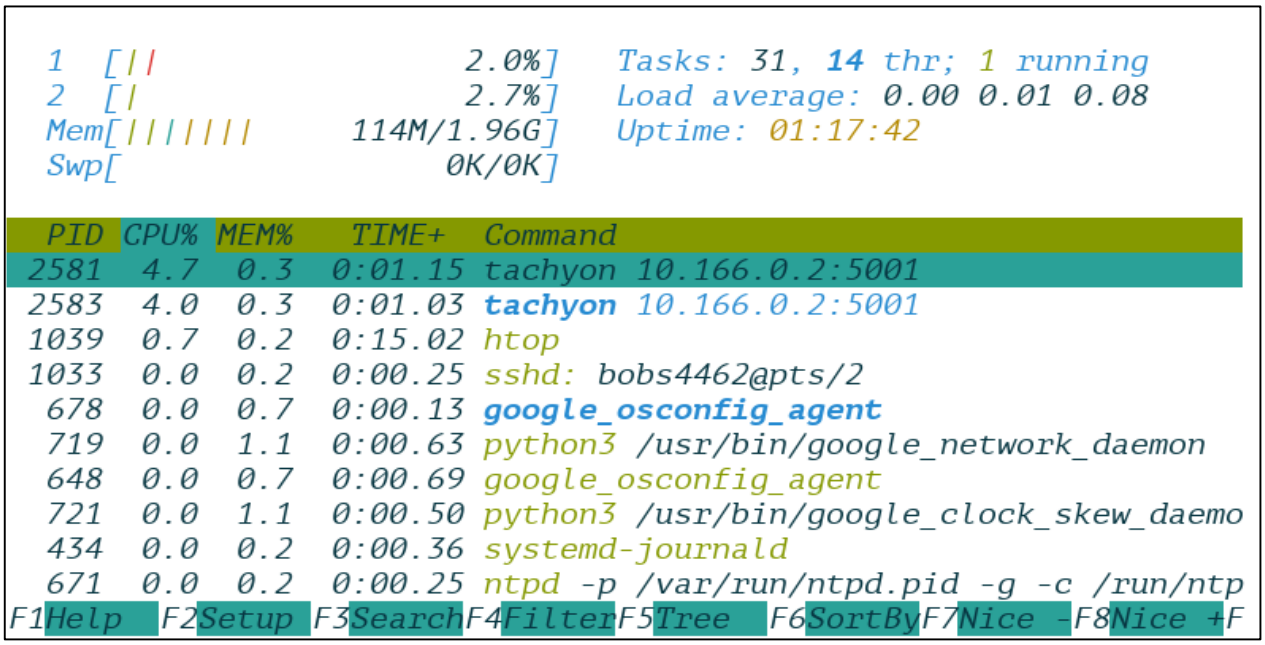

Рис. 3.5.3 Состояние системы, тест № 1, начало

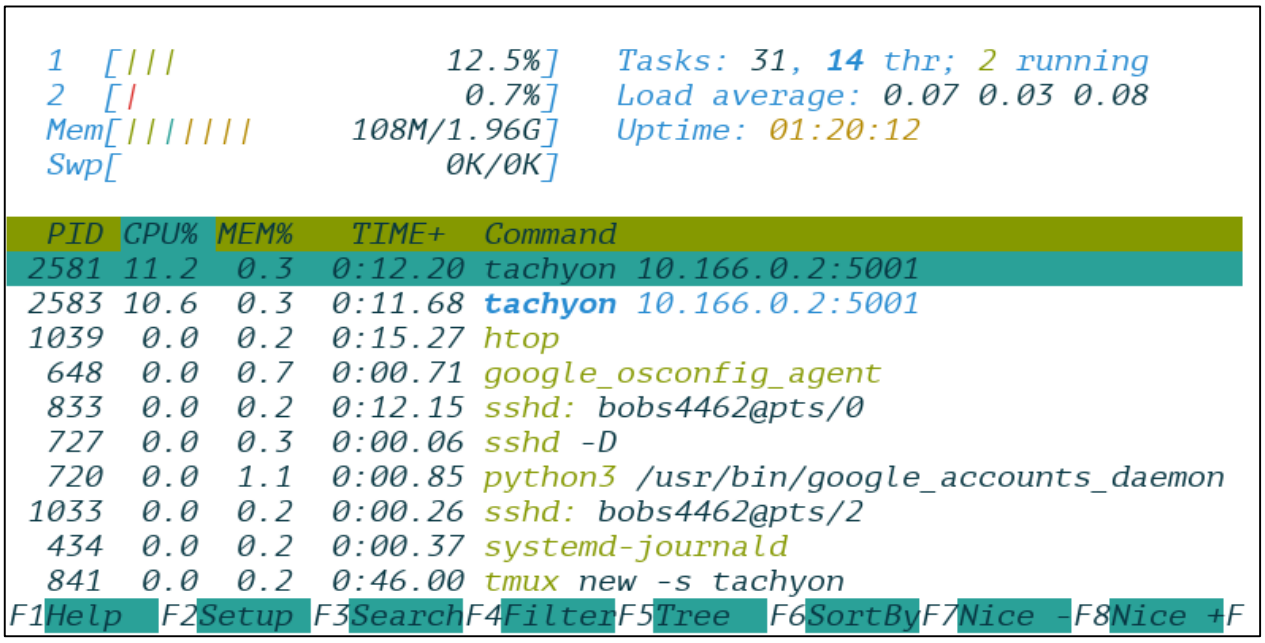

Рис. 3.5.4 Состояние системы, тест № 1, пиковая нагрузка

В связи с ограничением в пятьдесят одновременных подключений, для дальнейшего тестирования использовался другой сервис LoaderIO, который имеет ограничение по одновременным подключениям равным десяти тысячам. Соответственно второй тест производился с использованием ста новых подключений каждую секунду, что привело к ситуации, в которой одновременно веб-сервис обрабатывал двести запросов. Обращение к веб-сервису происходило из штата Вирджиния в США. Результат теста показан на Рис. 3.5.5.

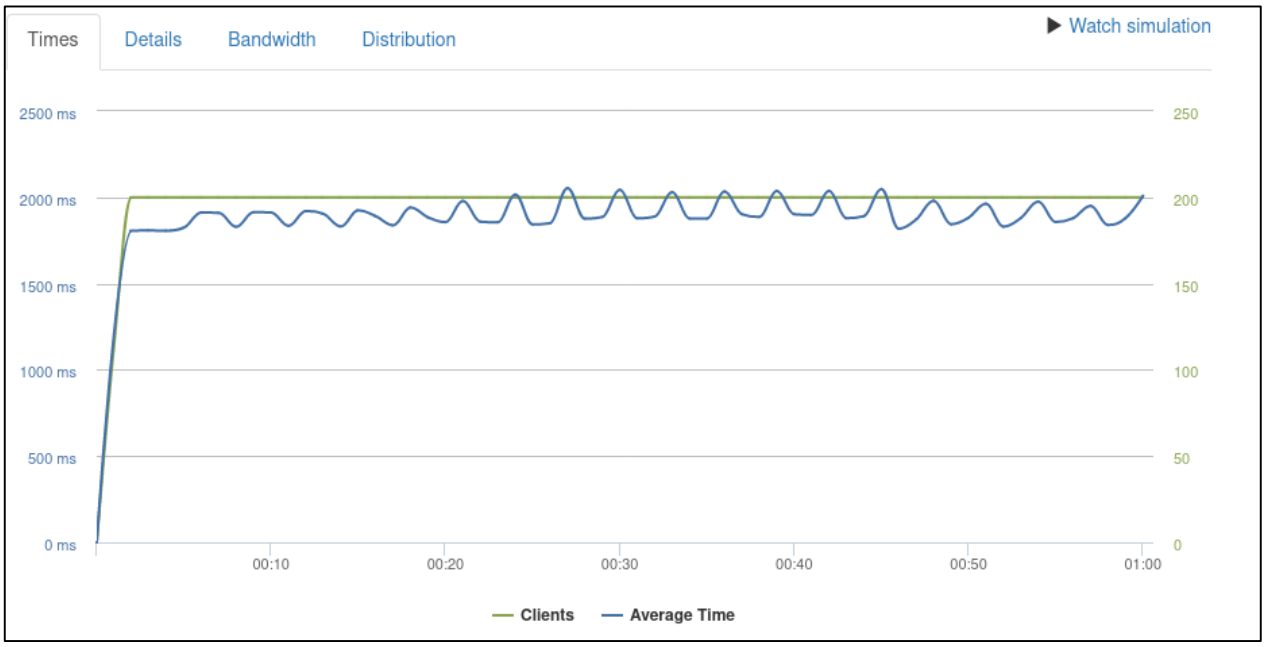

Рис. 3.5.5 Тест №2, 100 одновременных подключений

Время ответа сервиса колеблется в пределах двух тысяч миллисекунд, что по большей части связано с большой дистанцией между Вирджинией и Стокгольмом. Нагрузку на сервер можно увидеть на Рис. 3.5.6, по которому видно что в потребление процессорного времени веб-сервисом находится в пределах 20%.

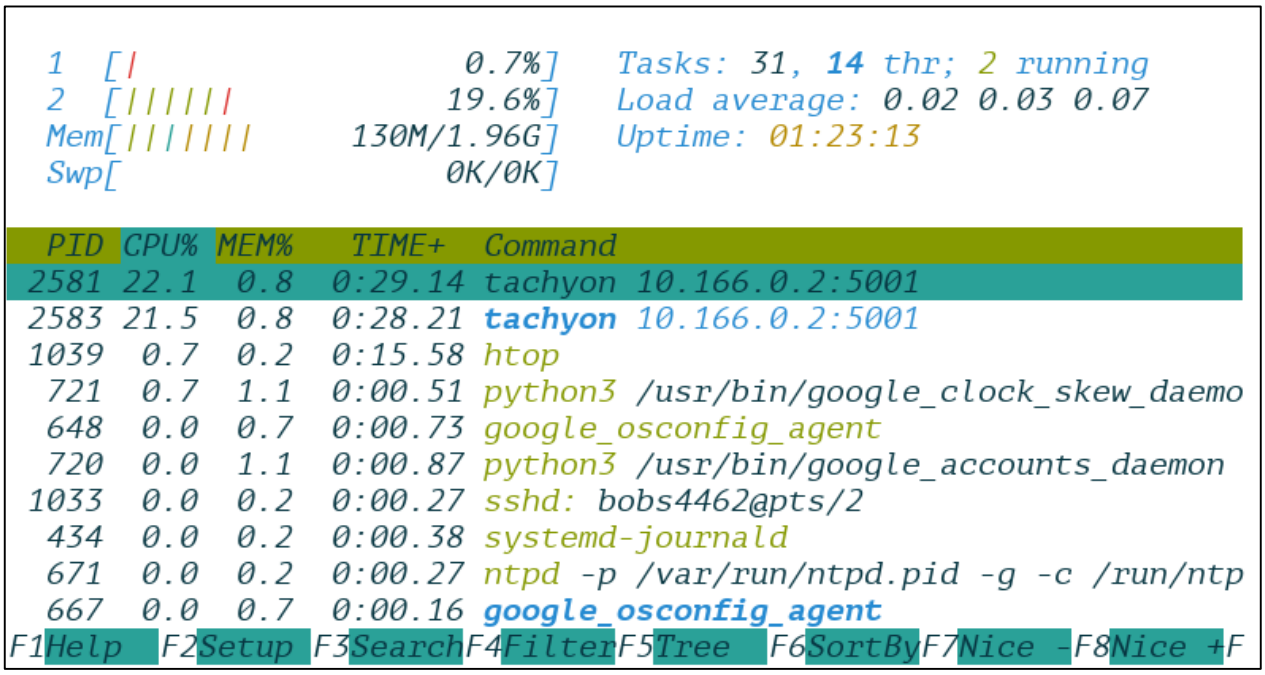

Рис. 3.5.6 Тест №2, состояние системы, пиковая нагрузка

Далее количество новых подключений в секунду было увеличено до двухсот. График времени ответа показан на Рис. 3.5.7. Время ответа сервиса чуть превысило две тысячи миллисекунд, а на нагрузка на систему изображена на Рис. 3.5.8. Потребление процессорного времени составило чуть более 50%.

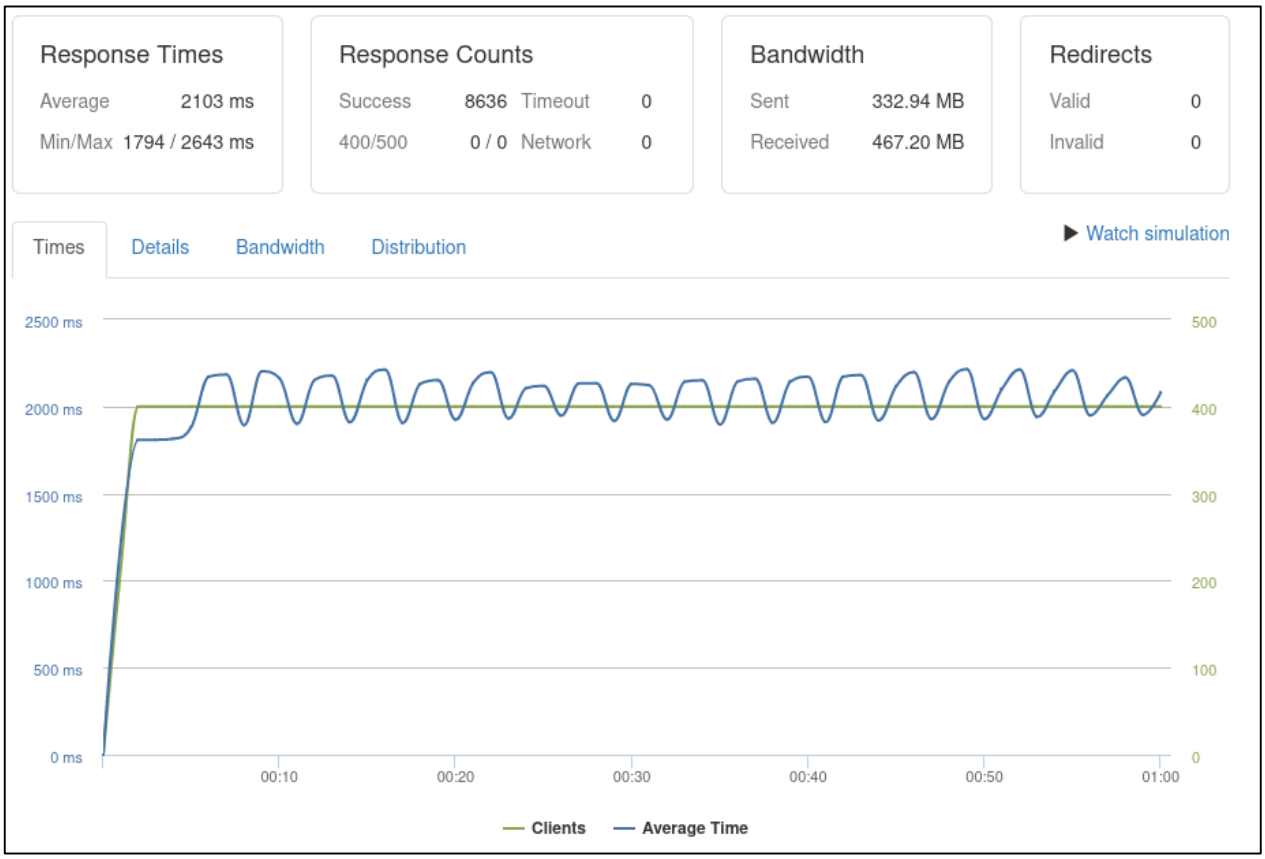

Рис. 3.5.7 Тест № 3, 200 одновременных подключений

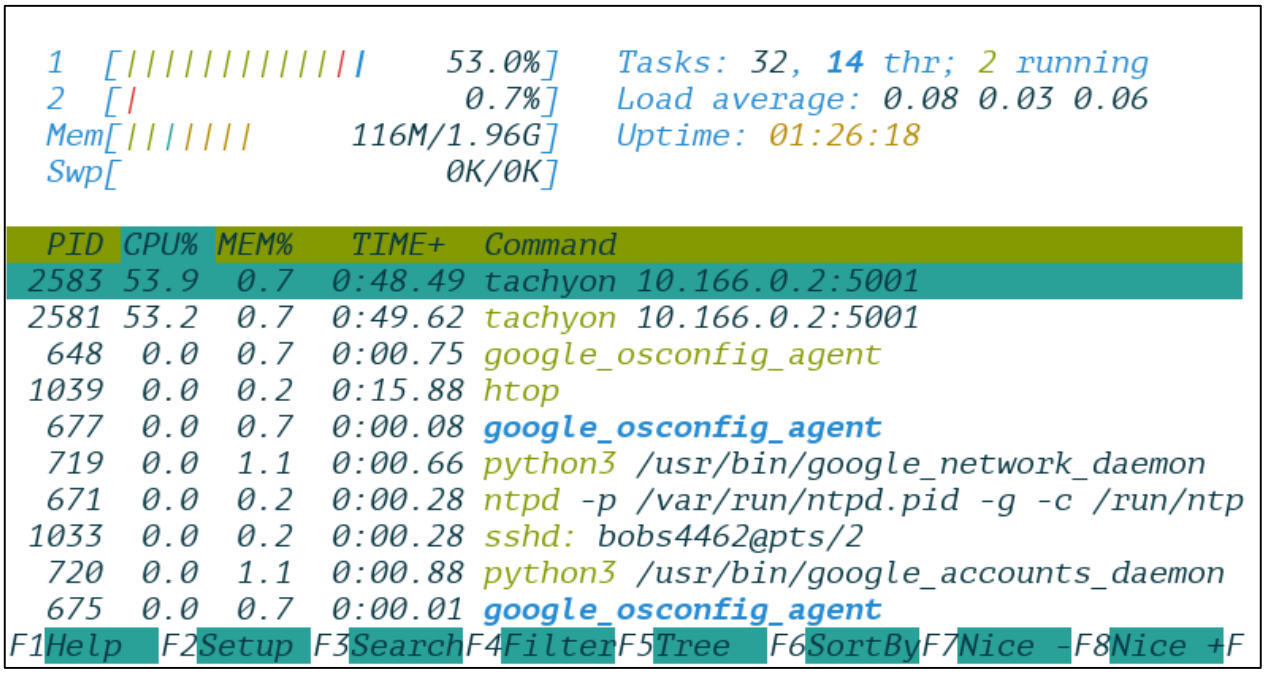

Рис. 3.5.8 Тест №3, состояние системы, пиковая нагрузка

Чтобы найти верхний предел возможностей веб-сервиса по обслуживанию запросов, количество одновременных подключений было увеличено до пятисот пятидесяти. График результатов показан на Рис. 3.5.9, из него видно что время ответа сервиса возросло до четырёх тысяч пятисот миллисекунд.

Рис. 3.5.9 Тест № 4, 550 одновременных подключений

Нагрузка на систему также возросла, до 100% процессорного времени, и полной загрузки одного из ядер процессора (Рис. 3.5.10).

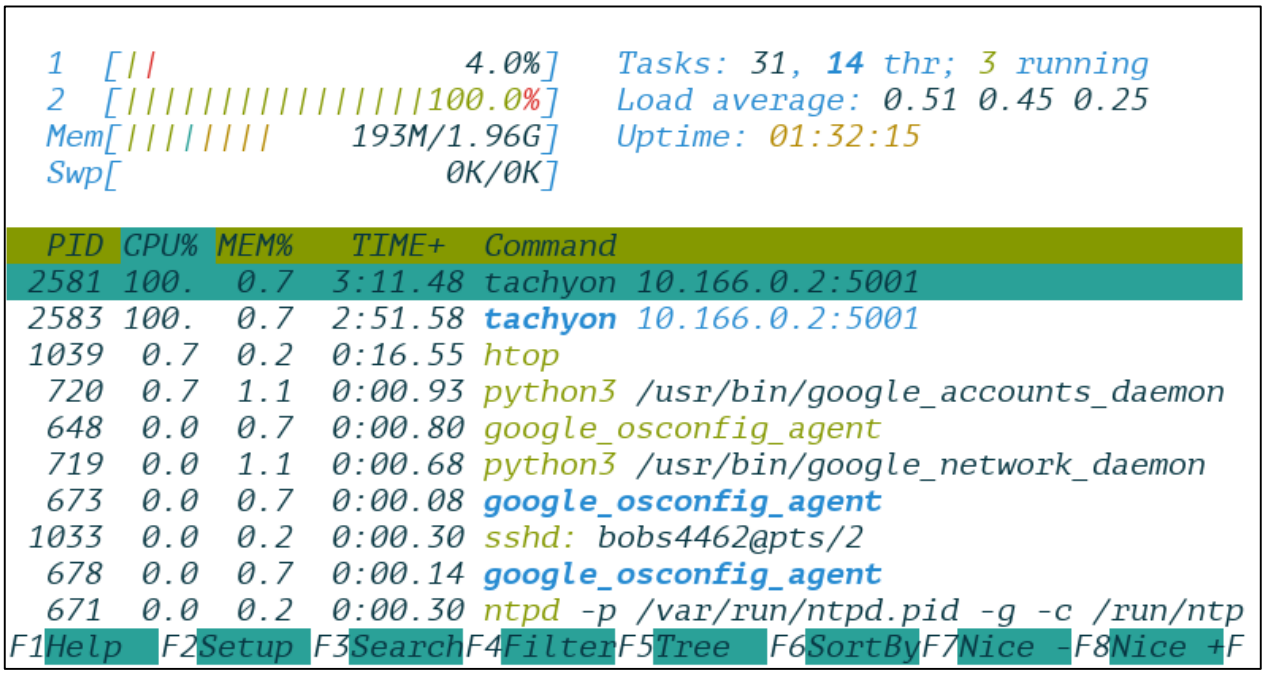

Рис. 3.5.10 Тест № 4, состояние системы, одно ядро полностью нагружено

Последним тестом было увеличение одновременных подключений до семисот. При этом произошёл резкий рост времени ответа до более чем двенадцать секунда на запрос (Рис. 3.5.11), и появлением ошибок в ответах, связанных с не способностью сервера обработать такое количество запросов. Загрузка процессора показана на Рис. 3.5.13, по нему видно что оба ядра процессора полностью нагружены. Возникновение ошибок связано с ограничением операционной системе на количество единовременно открытых файлов на процесс, которое по умолчанию составляет тысячу файлов. Данный лимит можно увеличить, но к снижению времени ответа это не приведёт ввиду того что процессор все равно не сможет обработать большее количество запросов.

По результатам тестов можно сделать вывод что разработанный веб-сервис обладает достаточно высокими показателями производительности, и может обрабатывать большое количество запросов одновременно. Эти показатели можно улучшить за счёт установки приложения на более

производительное аппаратное обеспечение и расположением клиентов в локальной сети.

Рис. 3.5.11 Тест № 5, 700 подключений, деградация производительности

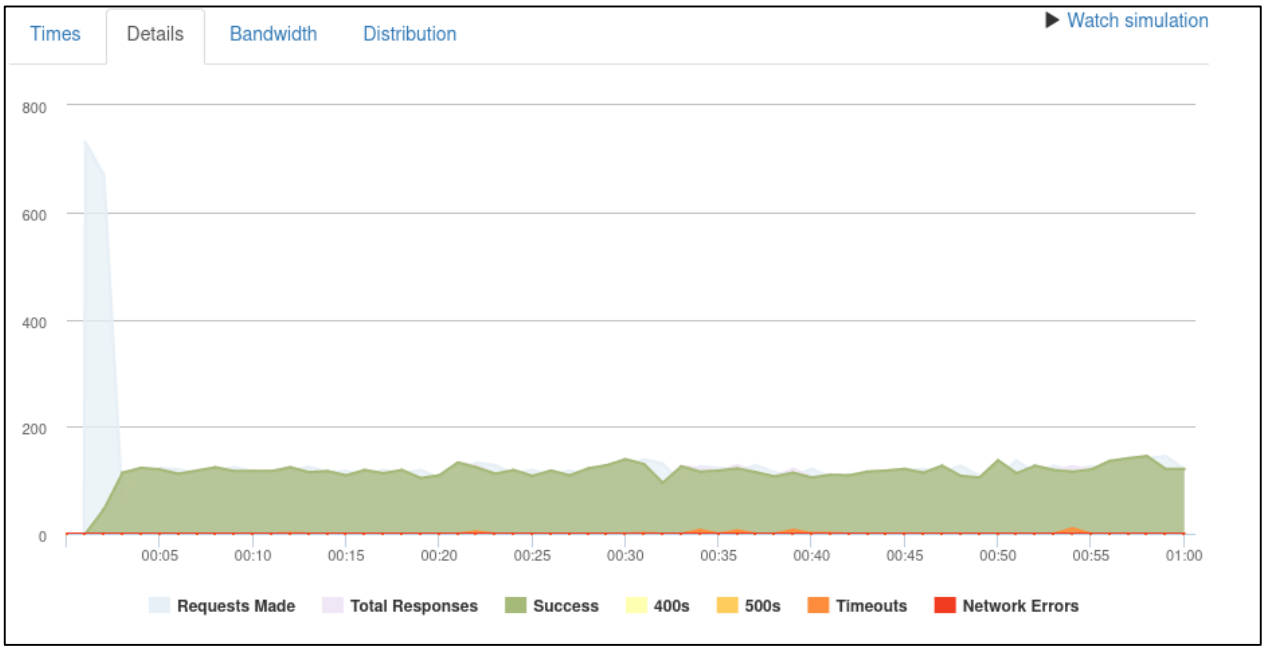

Рис. 3.5.12 Тест № 5, наличие ошибок в ответе от сервиса

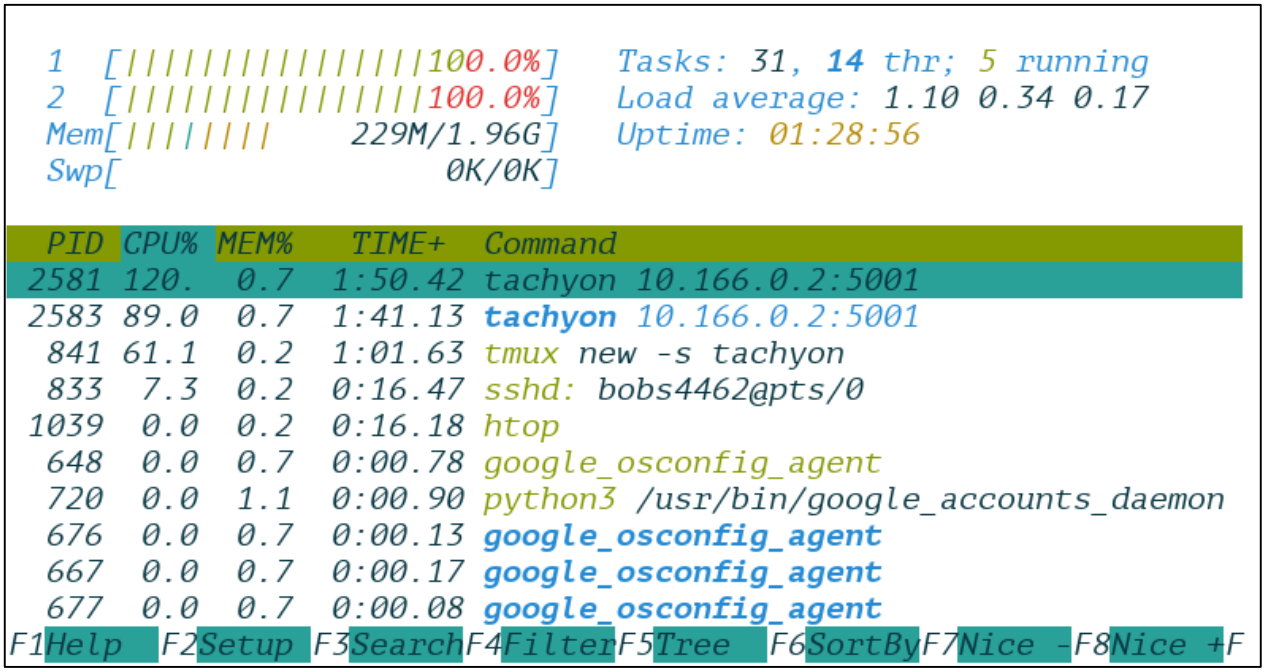

Рис. 3.5.13 Тест №5, состояние системы, оба ядра полностью нагружены

3.6. Потенциал к масштабированию

Масштабируемость – это способность системы справляться с увеличением рабочей нагрузки (увеличивать свою производительность) при

добавлении ресурсов (обычно аппаратных). Масштабируемость бывает вертикальной и горизонтальной. Вертикальная масштабируемость подразумевает увеличение производительности каждого компонента системы с целью повышения общей производительности. Масштабируемость в этом контексте означает возможность заменять в существующей вычислительной системе компоненты более мощными и быстрыми по мере роста требований и развития технологий. Горизонтальная масштабируемость представляет из себя разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам или увеличение количества серверов, параллельно выполняющих одну и ту же функцию. Масштабируемость в этом контексте означает возможность добавлять к системе новые узлы, серверы, процессоры для увеличения общей производительности.

Разработанная система шаблонизации хорошо масштабируется как вертикально, так и горизонтально. Но при вертикальном масштабировании лимитирующим фактором является пропускная способность сети. При горизонтальном масштабировании подобных лимитирующих факторов не имеется. Далее приведён один из способов горизонтального масштабирования.

В силу особенностей архитектуры разработанного приложения, можно легко развернуть несколько экземпляров веб-сервиса на разных вычислительных машинах, каждый из которых будет иметь доступ к централизованной базе веб-шаблонов. Так как изменение и добавление шаблонов является редкой по сравнению с чтением операцией, то обращений к этой базе будет значительно меньше, если правильно настроить механизм кеширования на каждом экземпляре веб-сервиса. Для распределения запросов между этими веб-сервисами можно настроить обратный-прокси сервер, например, на основе nginx. Такой подход позволит разворачивать

произвольное количество веб-сервисов в зависимости от нужд организации. После изменения базы веб-шаблонов одним из экземпляров веб-сервиса, он отправит остальным экземплярам запрос на обновление кеша, и те в свою очередь синхронизируют свой локальный кеш с базой веб-шаблонов. Схематически этот процесс изображен на Рис. 3.6.1.

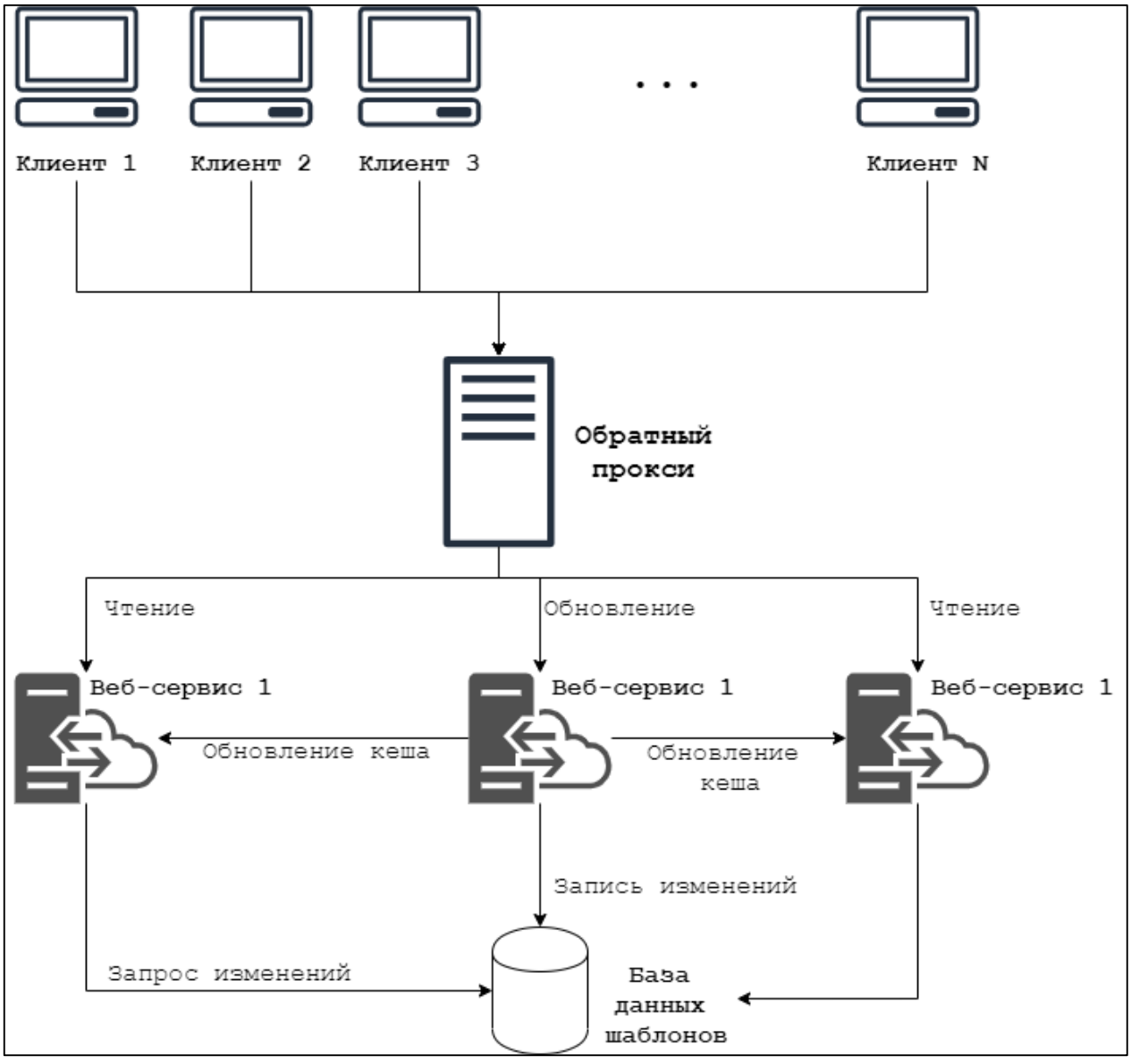

Рис. 3.6.1 Пример работы сервиса при горизонтальном масштабировании

Таким образом разработанный сервис особенно хорошо и легко поддаётся горизонтальному масштабированию, что позволит наращивать производительность, при возникновении такой необходимости.

3.7. Внедрение

Разработанная система шаблонизации была внедрена на предприятии ООО «Культурная служба», так же известной как билетный оператор «Пономиналу». В ИТ инфраструктуре компании в основном используется микро-сервисная архитектура для обеспечения работы служб компании. Такая архитектура предполагает наличие множества различных компонент, каждый из которых, вообще говоря может быть написан с использованием различных языков программирования. Эти компоненты могут формировать письма, например, для уведомления пользователя о сбое или наступлении какого-либо события. Компоненты, написанные на разных языках программирования, используют разные библиотеки шаблонизации, что приводило к необходимости поддерживать несколько разных продуктов, что является несколько затратным в отношении трудовых ресурсов.

С внедрением новой системы шаблонизации, все компоненты теперь могут обращаться к одному сервису, который имеет возможность обслуживать большое количество запросов одновременно. Этот факт значительно упрощает работы по сопровождению продуктов компании.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной выпускной квалификационной работы были изучены существующие решения для шаблонизации веб-документов. В ходе анализа были выявлены преимущества и недостатки, присущие самым популярным решениям. Также были идентифицированы проблемы, вызывающие снижение производительности рассмотренных систем веб-шаблонов.

На основе идентифицированных проблем был предложен и изучен ряд мер, направленных на повышение производительности систем веб-шаблонов. После чего предложенные способы повышения производительности были использованы для составления требований к выбору языка программирования и других инструментов для разработки высокопроизводительной системы шаблонизации.

Переда началом разработки было произведено моделирование работы системы шаблонизации с использование функциональных диаграмм в нотации IDEF0/DFD. После чего, с использованием языка программирования Rust и библиотек, существующих в экосистеме данного языка, был разработан асинхронный высокопроизводительный RESTful веб-сервис для шаблонизации, способный быстро обрабатывать большое количество одновременных запросов. Для удобства взаимодействия с данным веб-сервисом, к нему был разработан графический пользовательский интерфейс с применением JavaScript фреймворка Vue.js.

По окончанию разработки было произведено тестирование производительности веб-сервиса с направлением на него большого количества «тяжёлых» запросов одновременно. Результаты теста показали, что разработанный веб-сервис может одновременно справляться с очень

высокими нагрузками, даже не на очень производительном аппаратном обеспечении.

Разработанное приложение имеет детализированную документацию по применению, и уже может быть использовано на предприятиях для решения реальных задач.

СПИСОК ЛИТЕРАТУРЫ

1. ГОСТ Р ИСО/МЭК 15288-2005. Информационная технология. Системная инженерия. Процессы жизненного цикла систем.

2. ГОСТ Р ИСО/МЭК 25010-2015 Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов.

3. Машнин Тимур Сергеевич. Технология Web-сервисов платформы Java. – БХВ-Петербург, 2012. – С. 115. – 560 с.

4. Михаил Вюрш. Улучшение распознавания изменений в исходных кодах с помощью абстрактных синтаксических деревьев: дипломная работа [Текст]. Университет Цюриха 2006. – 64 с.

5. Twig Internals – Documentation [электронный ресурс]. Режим доступа: https://twig.symfony.com/doc/2.x/internals.html, свободный (дата обращения: 07.03.2020).

6. Chris Wanstrath. Mustache – Logic-less templates [электронный ресурс]: 2009. Режим доступа: https://mustache.github.io/mustache.5.html, свободный (дата обращения: 07.03.2020).

7. Flanagan, David. JavaScript: The Definitive Guide [Текст]. O'Reilly & Associates, 2006. – 992 с.

8. Introduction | Handlebars [электронный ресурс]. Режим доступа: https://handlebarsjs.com/guide/#what-is-handlebars, свободный (дата обращения: 07.03.2020).

9. How does memory allocation work in Python [электронный ресурс]. Режим доступа: https://medium.com/datadriveninvestor/how-does-memory-allocation-work-in-python-and-other-languages-d2d8a9398543, свободный (дата обращения: 14.03.2020).

10. Haskell and Rust [электронный ресурс]. Режим доступа: https://www.fpcomplete.com/blog/2018/11/haskell-and-rust, свободный (дата обращения 14.03.2020).

11. Блэнди Джим, Орендорф Джейсон. Программирование на языке Rust. – ДМК Пресс, 2018. – 550 с.

12. Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance – Computer, 1994. – 46 с.

13. Garbage collection and application performance | Dynatrace [электронный ресурс]. Режим доступа: https://www.dynatrace.com/resources/ebooks/javabook/impact-of-garbage-collection-on-performance/, свободный (дата обращения 14.03.2020).

14. Леонард Ричардсон, Сэм Руби. RESTful веб-сервисы, O'Reilly Media, 2007. – 454 с.

15. Rust async book [электронный ресурс]. Режим доступа: https://rust-lang.github.io/async-book/, свободный (дата обращения 15.04.2020).

16. How a template engine works – Shipeng Feng's writings [электронный ресурс]. Режим доступа: https://fengsp.github.io/blog/2016/8/how-a-template-engine-works/, свободный (дата обращения 25.03.2020).

17.

ПРИЛОЖЕНИЕ А.

ДОКУМЕНТАЦИЯ API

Получение шаблона без форматирования:

Метод: /raw/{ название файла шаблона }
Тип запроса: GET
Параметры: отсутствуют
Тип ответа: строковый
Содержимое ответа: запрошенный шаблон
Возможные ошибки:

- **Код: 404**. Причина: запрошенный шаблона не существует.

Получение сгенерированного документа:

Метод: /render/{ название файла шаблона }

Тип запроса: POST

Параметры: JSON – источник данных для шаблонизации

Тип ответа: строковый

Содержимое ответа: сгенерированный веб-документ

Возможные ошибки:

- **Код: 404**. Причина: запрошенный шаблон не существует;
- **Код: 501.** Причина: ошибка валидации данных.

Обновление существующего шаблона:

Метод: /update/{ название файла шаблона }

Тип запроса: POST

Параметры: обновлённый текст шаблона

Тип ответа: строковый

Содержимое ответа: ОК в случае успеха

Возможные ошибки:

- **Код: 404**. Причина: запрошенный шаблон не существует;
- **Код: 503.** Причина: ошибка валидации шаблона.

Добавление нового шаблона:

Метод: /add/{ название файла шаблона }

Тип запроса: POST

Параметры: текст нового шаблона

Тип ответа: строковый

Содержимое ответа: OK в случае успеха

Возможные ошибки:

- **Код: 503.** Причина: ошибка валидации шаблона;
- **Код: 504.** Причина: ошибка записи шаблона в хранилище.

Удаление существующего шаблона:

Метод: /remove/{ название файла шаблона }

Тип запроса: POST

Параметры: название фала шаблона

Тип ответа: строковый

Содержимое ответа: OK в случае успеха

Возможные ошибки:

- **Код: 404**. Причина: запрошенный шаблон не существует;
- **Код: 505.** Причина: ошибка удаления шаблона из хранилища.

ПРИЛОЖЕНИЕ Б.

СИНТАКСИС ДЛЯ НАПИСАНИЯ ШАБЛОНОВ

Основы

Любой шаблон является просто текстовым файлом, в котором переменные и выражения будут заменены значениями, при отрисовке (рендере). Синтаксис основан на шаблонах Jinja2 и Django.
Существует 3 вида разделителей, которые нельзя изменить:

- `{{` и `}}` для выражений
- `{%` или `{%-` и `%}` или `-%}` для операторов
- `{#` и `#}` для комментариев

Без отрисовки

Весь текст внутри блока raw как строку и не будет пытаться отобразить то, что внутри. Полезно, если у вас есть текст, содержащий разделители.

```
{%raw %}

Здравствуйте {{name}}

{%endraw%}
```

будет отображаться как `Здравствуйте {{name}}`.

Контроль пробельных символов

Tera поставляется с простым в использовании управлением пробелами: используйте {%- если вы хотите удалить все пробелы перед оператором и -%} если вы хотите удалить все пробелы после.
Например, давайте посмотрим на следующий шаблон:

```
{% set my_var = 2%}
{{my_var}}
```

будет иметь следующий вывод:

```
2

```

Если мы хотим избавиться от пустой строки, мы можем написать следующее:

```
{% set my_var = 2 -%}
{{my_var}}
```

Комментарии

Для комментариев в шаблонах поместите текст между {# и #}.

```
{# Это комментарий #}
```

Литералы
Тера имеет следующие литералы

- **Булевы:** true или false
- **Целочисленные**
- **С плавающей точкой**
- **Строки:** текст находищийся между двойными или одинарными ковычками
- **Массивы:** литералы находищийся `[` и `]` и разделенных запятыми

Переменные

Для отображения переменных контекста необходимо обернуть нужную переменную между `{{` и `}}`
Если попытаться отобразить переменную которая не существует, то это приведёт к ошибке построения документа

Доступ к членам структуры данных
Для получения доступе к члену структуры данных существует 2 способа

1. **Точечная нотация:** например

   ```
   {{ product.name }}
   ```

2. **Нотация квадратных скобок:** например

   ```
   {{ product['name'] }}
   ```

Для получения доступа к элементам массива также можно использовать обе нотации, где после точки или внутри квадратных скобок должен быть индекс элемента

Выражения
Выражения можно использовать почти везде, существует несколько видов выражений:

- Математические:сложение, вычитание, умножение, деление, деление по модулю

  Операндами этих опереций могут быть только числа

- Сравнения
- Логические: and, or, not

- Соеденение строк: используя оператор конкатенации ~, например

  `{{ product.name ~ " is the best " in city product.city }}`

- Вхождение во множество: используя оператор вхождения in, например

  `{{ product.name in ["cola", "sprite", "fanta"] }}`