**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ. ПОСТАВНОКА ЗАДАЧИ

**ВВЕДЕНИЕ**

После разработки и развертывания веб-сайтов возникает необходимость контроля их работоспособности с дальнейшей целью её поддержания. Это включает в себя как проверку доступности самого сервера (т.н. uptime), так и проверку работоспособности отдельных конечных точек веб-интерфейса, отвечающих за различные бизнес-процессы, будь то, например, регистрация пользователя или оформление заказа.

Например, модуль управления аккаунтами, как правило, говоря в общих терминах, представляется отдельными классами, а говоря в терминах ASP.NET – отдельным контроллером, для которого возможно разработка изолированных сценариев тестирования. Такой сценарий может включать, как вариант, следующие шаги:

1. Создание нового аккаунта.
2. Подтверждение почты.
3. Смена пароля.
4. Удаление аккаунта.

В разрабатываемом проекте предлагается реализовать графический редактор сценариев запросов. Запрос, в общем случае, представляет собой адрес запроса и передаваемую информацию. В отдельных случаях он может представлять собой получение некоторой строки из письма на электронной почте. В редакторе предлагается реализовать возможность обработки ответа путем ввода регулярного выражения. Если ответ приходит в формате JSON, то предлагается обращаться к нему напрямую через переменные. Извлеченные из ответа переменные предлагается записывать в переменные сценария, что обеспечивает дальнейшее использование. В данном случае остро встаёт вопрос безопасности, так как, по сути, предлагается реализовать функционал исполнения произвольного кода. В этом случае требуется либо разворачивать JavaScript Runtime в отдельном окружении, либо изолировать его каким-либо другим образом.

Дополнительной возможностью предполагается отправка запросов с серверов, расположенных в других локациях. Это может быть реализовано как развертыванием вспомогательных нод сервиса, которые бы представляли из себя полноценные HTTP сервера, так и отправкой запроса через развернутые или сторонние (что увеличивает количество доступных локаций на порядки) SOCKS5 прокси-сервера, что обеспечивает тот же результат, но значительно проще в реализации.

Также кажется реалистичной реализация валидации email-адреса, т.е. реализация возможности написания сложных сценариев, включающих взаимодействие с почтой через протоколы IMAP/POP3, извлечением кода подтверждения аккаунта, например, при тестировании регистрации, с дальнейшей передачей его в запросе.

Но наиболее сложной задачей, на мой взгляд, в предлагаемой разработке, является задача синхронизации работы компонентов – сохраненных в памяти сценариев, экземпляров данного сценария, которые сейчас в прогрессе, прогресс каждого отдельного действия в данном сценарии, и, в конце концов, непосредственно выполняемой для этого действия работы, ведь если действие – это запрос данных с извлечением результата, то можно легко понять, что на уровне кода это представляет вовсе не одну строку...

Очевидно, что выполнение т.н. сценариев, в связи с их длительностью, не может быть реализовано в блокирующем режиме, но и в асинхронном режиме, предстоит решить проблему сохранение прогресса работы сценария.

Универсальность решения должна позволить внедрить его как в существующие, так и в новые системы.

**КОНЦЕПТУАЛЬНОЕ ПРОЕКТИРОВАНИЕ**

**«**Управление сложностью – самый важный технический аспект разработки ПО. По-моему, управление сложностью настолько важно, что оно должно быть Главным Техническим Императивом Разработки ПО.»

- Стив Макконнелл, «Совершенный код»

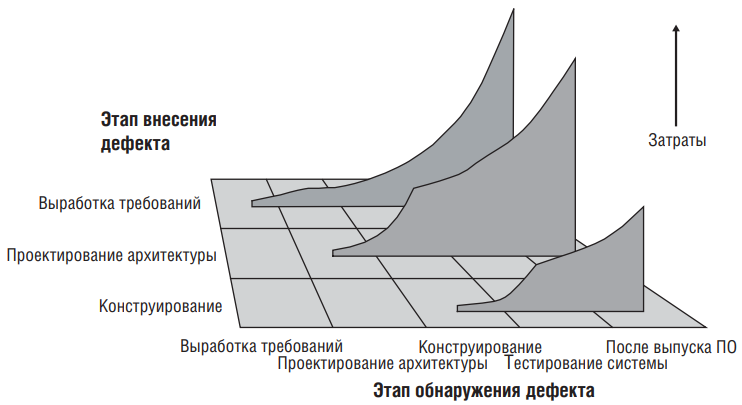
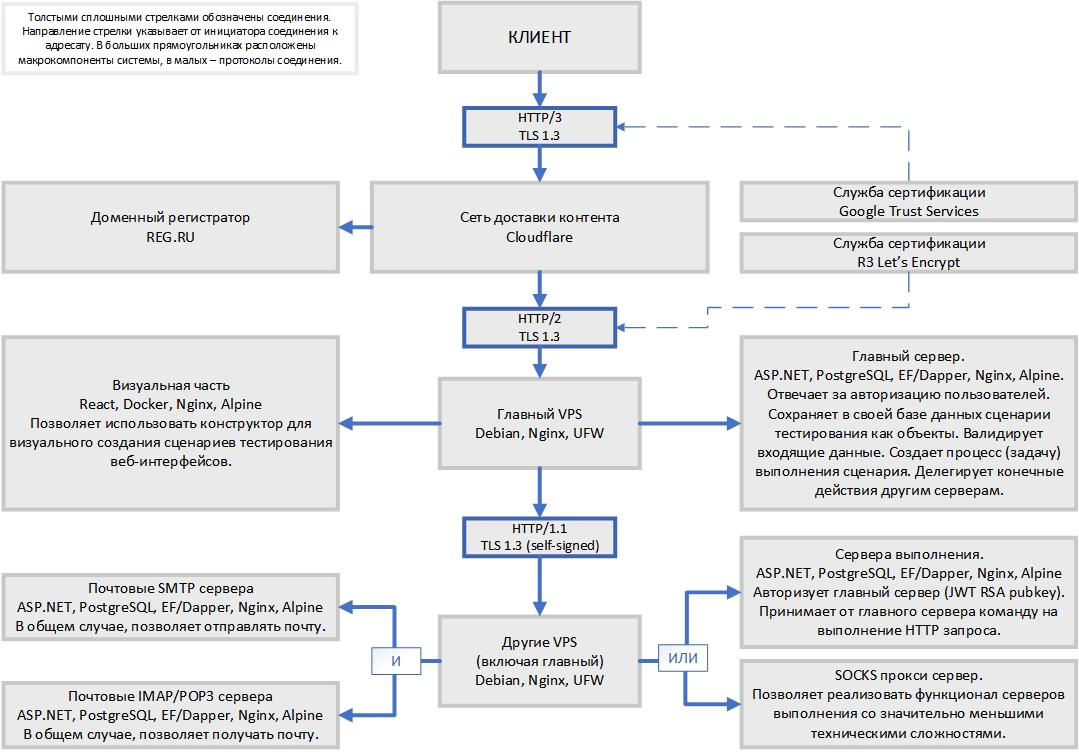
Выполнить высокоуровневое проектирование необходимо перед переходом к проектированию отдельных компонентов. Выполнение данного этапа позволяет чётко понять, какие интерфейсы должны иметь те или иные части системы. Ошибка же на этапе проектирования архитектуры, как показано на рисунке 1, влечет существенное увеличение затрат на разработку ПО в будущем.

Рисунок 1 (С. Макконнелл, «Совершенный код») - c увеличением интервала между моментами внесения и обнаружения дефекта стоимость его исправления сильно возрастает. Это верно и для очень последовательных проектов (выработка требований и проектирование на 100% выполняются заблаговременно), и для очень итеративных (аналогичный показатель равен 5%).

На блок-схеме 1 представлена концептуальная схема. Как видно, имеется явное разделение бизнес-задач между подсистемами. Согласно нескольким докладам конференции «HighLoad++», разделение подсистем должно осуществляться именно по бизнес-задачам, а не по стеку технологий. В противном случае вы не получаете никаких преимуществ микросервисной архитектуры, но при этом получается всё её минусы. Т.е. само по себе распределение по разным серверам (включая как физические, так и виртуальные и даже программные сервера!) систем, решающих одну задачу, является фактором, усложняющим систему. Напротив, разделение систем по решаемым задачам с созданием при этом чёткого и просто интерфейса решает проблему инкапсуляции сложности каждой отдельной задачи от всей остальной системы. Такая архитектура позволяет быстро создать законченные микросистемы, при этом сохранить их простоту.

Разделение систем по интерфейсам крайне важно, поскольку обеспечивает их раздельное версионирование. Так, можно разрабатывать новую версию API для главного сервера, при этом совершенно не затрагивая клиент, не порождая необходимость его пересборки и развертывания.



Блок-схема 1 – концептуальная схема системы

**ВЫБОР ТЕХНОЛОГИЧЕСКОГО СТЕКА**

«На улицах ясно одно: важнее всего скорость разработки.»

- Cедат Капаноглу, «Кодер с улицы»

По аналогии с тем, что «самый безопасный способ хранить данные – не хранить данные», я убежден, что самый лучший способ написать хороший код – не писать его. Если отойти от лирики, то разработка должна сводится к использованию набора готовых решений. Человек, реализующий каждый алгоритм с нуля, не продвинется дальше написания калькулятора. Время программиста в современном мире стоит значительно больше времени процессора, способного производить миллиарды вычислений в секунду. Однако это не значит, что нужно как можно быстрее писать плохой код, речь идет только о сокращении необходимости это делать:

«Главный Закон Контроля Качества ПО заключается в том, что повышение качества системы снижает расходы на ее разработку.»

- Стив Макконнелл, «Совершенный код»

В противном случае, т.н. технический долг рано или поздно приведет к невозможности развивать проект оставаясь в рамках разумных трудозатрат.

Исходя из вышесказанного, очевидными становятся два критерия выбора в пользу той или иной технологии:

1. Большое комьюнити, а значит – большое количество готовых библиотек.
2. Наличие у разработчика опыта работы с конкретной технологией.

**СОЗДАНИЕ СТРУКТУРЫ ИСХОДНОГО КОДА**

Перед непосредственно разработкой, очевидно, необходимо создать некоторую будущую структуру. Для начала, это банально сводится к созданию четырех папок: для клиента, серверной части, imap-микросервиса, документации.

В программной коде 1 представлен PowerShell-скрипт, вызываемый для создания шаблона на примере главного сервера. В данном случае создается globaljson файл, в котором указывается версия dotnet SDK общая, для всех проектов в папке с ним. Далее создается два проекта – webapi и xunit, где первый – для работы с API посредствам фреймворка ASP, второй – для выполнения тестирования проектов решения. Файл gitignore служит для выбора файлов, игнорируемых системой контроля версий git. Файл dockerfile служит как файл сценария сборки docker-образа, что будет рассмотрено позднее. Файл docker-compose и его производные служат как файлы сценария запуска готов docker-образов. Файл dockerignore служит аналогичным целям, что и файл gitignore, но, как очевидно, для другого программного обеспечения.

Программный код 1 – PowerShell-скрипт создания шаблона проекта главного сервера

dotnet new globaljson;

dotnet new webapi -o wtt\_main\_server\_api;  
dotnet new xunit -o wtt\_main\_server\_tests;

dotnet new sln --name wtt\_main\_server;  
dotnet sln add wtt\_main\_server\_api/wtt\_main\_server\_api.csproj;  
dotnet sln add wtt\_main\_server\_tests/wtt\_main\_server\_tests.csproj;

dotnet new gitignore;

New-Item dockerfile;  
New-Item docker-compose.yml;  
New-Item docker-compose.override.yml;  
New-Item .dockerignore;  
Set-Content .dockerignore '...';

git init;

Для создания шаблона приложения веб-интерфейса применим наиболее простой метод – встроенную npm-команду, которая помимо создания React-приложения на языке TypeScript, также инициализирует git-репозиторий: программный код 2.

Программный код 2 – команда создания React-приложения на языке TypeScript

npx create-react-app my-app --template typescript

После выполненных шагов создаются удаленные репозитории. В моём случае, я использую сервис github. Далее, как представлено в программном коде 3, на примере главного сервера, осуществляется загрузка данных.

Программный код 3 – набор команд для загрузки данных на удаленный git-сервер (github)

git remote add origin https://github.com/LuminoDiode/wtt\_main\_server;  
git add \*;  
git commit -m “init”;  
git push origin master;

ИСТОЧНИКИ  
1. https://www.youtube.com/watch?v=eI1QQUrFUZI