Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б15-мм

Автоматическая генерация интеграционных тестов для веб приложений на платформе ASP.NET

Алексеев Артур Игоревич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
доцент кафедры СП, к. ф.-м. н., Д. А. Мордвинов

Консультант:
инженер-исследователь, М. П. Костицын

Санкт-Петербург
2024

Оглавление

Введение

Тестирование приложений является неотъемлемой частью процесса разработки, которая позволяет выявить недостатки и ошибки, а так же улучшить надежность и безопасность продукта. Однако, писать тесты — это большая и трудная задача, которая может отнять много времени у программиста. Также, из-за человеческой ошибки, не все ситуации в программе легко предугадать, и при тестировании приложения можно не увидеть важные ошибки и уязвимости. Символьное исполнение представляет собой технику анализа кода приложения, которая позволяет найти все наборы входных данных для выполнения всех возможных путей. Эта техника автоматизирует процесс создания тестов, а так же гарантирует, что ни один случай не будет пропущен.

С появлением все большего числа веб-приложений и ростом их роли в нашей повседневной жизни, необходимость в их тестировании становится все более актуальной. Помимо улучшения опыта использования, появится возможность выявлять критические уязвимости приложений, которые могут использоваться злоумышленниками. В контексте веб-приложений особенно важно интеграционное тестирование, которое позволяет проверить работу приложения в целом, включая взаимодействие различных компонентов и систем.

Для создания веб-приложений можно использовать различные инструменты, в том числе платформу .NET[1] и ASP.NET Core[5]. Они предоставляют широкие возможности для разработки и тестирования приложений, а также обладают высокой производительностью и масштабируемостью. ASP.NET Core также имеет открытый исходный код, и является кроссплатформенным.

Для автоматической генерации тестов в .NET существует инструмент V# [10]. V# — это движок символьного исполнения для программ на платформе .NET. Он использует символьное исполнение для создания тестов, которые покрывают все возможные пути выполнения приложения. С помощью V# уже можно генерировать модульные тесты для веб-приложений, но исследовать приложение полностью, от полу-

чения запроса до его обработки и возвращения ответа пользователю, не представляется возможным. Наша задача - расширить возможности символьного движка V# и исследовать с его помощью веб-приложения на основе ASP.NET Core.

Дата сборки: 3 июня 2024 г.

1. Постановка задачи

Целью данной работы является автоматическая генерация тестов для веб-приложений на платформе ASP.NET Core с помощью V#. Для достижения цели были сформулированы следующие задачи.

1. Выполнить обзор литературы о существующих решениях по генерации тестов для веб-приложений.
2. Создать алгоритм для генерации интеграционных тестов для веб-приложений на основе ASP.NET Core.
3. Спроектировать архитектуру решения.
4. Реализовать спроектированное решение в проекте V#.
5. Провести тестирование.

2. Обзор

В данной главе описывается общий алгоритм работы механизма символьного исполнения, приводятся некоторые примеры приложений на ASP.NET, а так же рассматривается написание интеграционных тестов с использованием WebApplicationFactory.

2.1. Символьное исполнение

Символьное исполнение представляет собой технику анализа программного обеспечения, которая позволяет найти все наборы входных данных, способствующие выполнению каждого из его возможных путей. Вместо того чтобы запускать программу на конкретных входных данных, символьное исполнение работает с символьными значениями, которые представляют переменные и память в программе.

Процесс символьного исполнения начинается с создания символьных значений для всех входных данных программы. Затем анализатор программы строит граф потока управления, который показывает все возможные пути выполнения программы. Для каждого пути выполнения анализатор выполняет символьное вычисление значений переменных на каждом шаге выполнения программы.

Важным понятием в символьном исполнении является символьное состояние, которое отражает состояние программы в конкретной точке исполнения. Частями символьного состояния являются

1. Инструкция — инструкция, которая исполняется в данный момент

2. Ограничения условия пути π — множество ограничений на символьные значения переменных, при которых поток исполнения дойдет до данной точки. Например, если до этого встречалась конструкция if(a $>$ 10), и мы вышли в ветку then, тогда в ограничении условия пути будет условие a $>$ 10.

3. Символьная память — отображение из множества объектов программы в множество их символьных значений.

Символьная машина выполняет инструкции программы, обновляя инструкцию, ограничения условия пути и память у состояний. В случае, когда исполнение программы может пойти не одним путем, создаются новые состояния, с соответствующими значениями ограничения условия пути.

После того как символьное исполнение завершено для всех возможных путей выполнения программы, анализатор может использовать полученную информацию для выявления потенциальных ошибок и уязвимостей, или для создания тестов с большим уровнем покрытия. Например, если в результате работы символьного движка получился ряд состояний, можно подобрать параметры, подходящие под ограничения условия пути, и использовать их как входные данные для тестов, которые покроют все возможные случаи.

2.2. Пример работы символьного движка

Листинг 1: Функция target

```
bool target(int x, int y){
x = 4;
if (x > y * 2) return true;
if (x < y) return false;
error();
}
```

Для исследования символьным движком функции target, создается начальное состояние, в котором инструкция — первая инструкция в функции, а ограничение условия пути — true. Далее, создается и меняется ряд состояний

1. При исполнении первой инструкции, x = 4, в символьной памяти переменная x принимает значение 4

2. После этого исполняется инструкция if(x > y * 2), которая порождает 2 состояния.

3. Одно находится в ветке then, с ограничением пути, удовлетворяющем условию x > y * 2, $\pi = 4 > \alpha_y * 2$ и на инструкции return true, после исполнения которой результат функции target будет true

4. Другое состояние находится в ветке else, на инструкции if (x < y), с ограничением пути !(x > y * 2), то есть $\pi = !(4 > \alpha_y * 2)$. При исполнении данной инструкции появится еще 2 состояния

5. Одно находится в ветке then, с ограничением пути, удовлетворяющем условию x > y * 2 && x < y, $\pi = !(4 > \alpha_y * 2)$ && $4 < \alpha_y$ и на инструкции return false, после исполнения которой результат функции target будет false

6. Другое находится в ветке else, с ограничением пути $\pi = !(4 > \alpha_y * 2)$ && $!(4 < \alpha_y)$, на инструкции error(), после исполнения которой результатом функции target будет ошибка

На основе трех получившихся финальных состояний (3, 5, 6), можно сделать вывод о поведении функции при различных входных данных, и создать тесты. Но для этого нужно узнать конкретные значения переменных, которые удовлетворяют формуле, заданной в ограничениях условия пути. Задача поиска значений символьных переменных, при которых условие пути выполняется, может быть представлена как задача выполнимости в формулы в теориях, и для решения этой задачи используются SMT решатели. SMT решатели могут найти конкретные значения символьных переменных для заданного условия, например, для 3 состояния решением будет x = 0, y = 1, для второго — x = 0, y = 5, а для состояния 6 — x = 0, y = 3. Схема символьного исполнения представлена на Рис. 1.

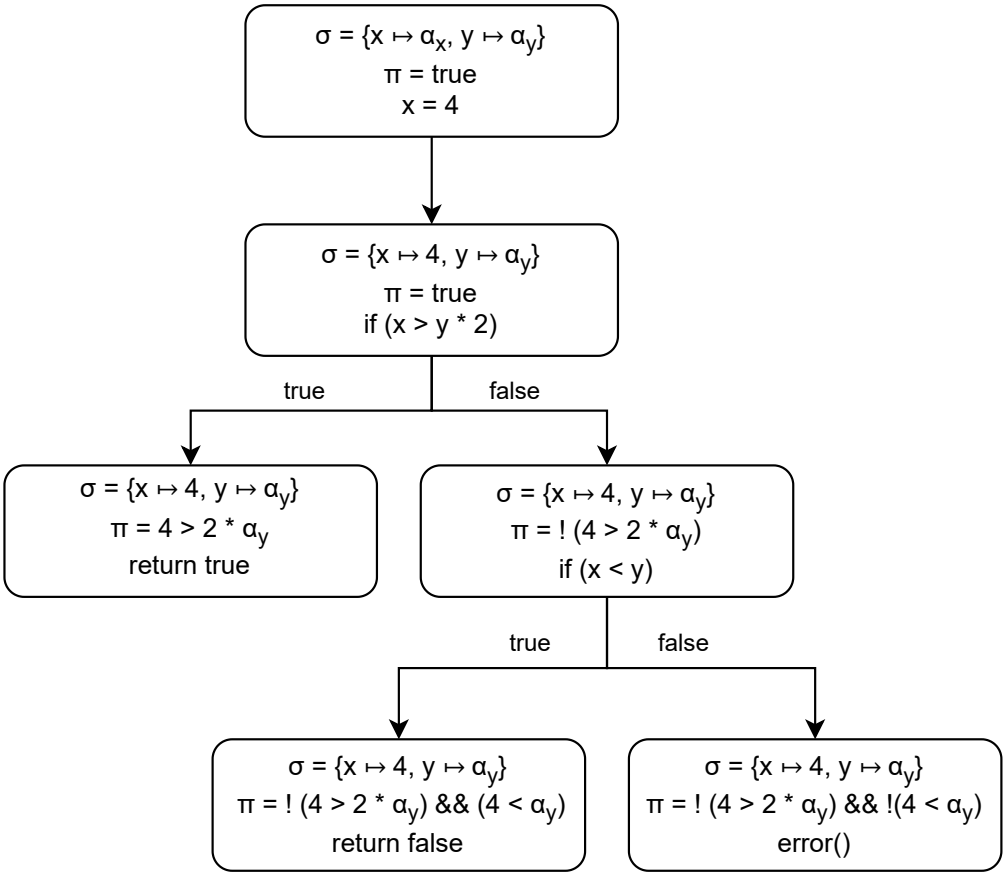

Рис. 1: Схема символьного исполнения для функции target

2.3. Веб-программирование на платформе .NET

.NET[1] — это платформа для разработки приложений, которая поддерживает несколько языков программирования, включая C#, F# и VB.NET. Она предоставляет множество инструментов для разработки приложений, включая среду разработки Visual Studio, библиотеки классов .NET и множество других инструментов. Она также поддерживает множество платформ, включая Windows, Linux и macOS.

Для создания веб-приложений в .NET имеется ряд фреймворков, таких, как

1. ASP.NET Core
2. Coalesce
3. FubuMVC

4. NancyFx

5. IISNode

6. Giraffe

ASP.NET Core[5] — это фреймворк для веб-разработки, который основан на платформе .NET, разработанный компанией Microsoft и предоставляет множество инструментов и библиотек для разработки веб-приложений, включая поддержку MVC (Model-View-Controller), Web API, SignalR и многое другое. Он также предоставляет возможность разработки кроссплатформенных приложений, которые могут работать на Windows, Linux и macOS. Согласно исследованию[3], проведенному сайтом Stack Overflow в 2023 году, ASP.NET Core был назван самым популярным веб-фреймворком среди разработчиков на платформе .NET.

2.4. Структура проектов на основе ASP.NET Core

ASP.NET Core является гибким фреймворком, который позволяет создавать проекты с разной структурой и архитектурой. Например

1. MVC (Model-View-Controller) — это структура проекта, которая предполагает наличия моделей (предоставляет данные и реагирует на команды контроллера, изменяя своё состояние), представление (отвечает за отображение данных модели пользователю, реагируя на изменения модели) и контроллер (интерпретирует действия пользователя, оповещая модель о необходимости изменений).

2. Web API — это структура проекта, которая предназначена для создания веб-сервисов и API. Она не содержит представлений и предоставляет только контроллеры и модели, которые могут возвращать данные в формате JSON или XML.

3. Razor Pages — это структура проекта, которая предоставляет более простой подход к созданию веб-страниц. Она использует Razor-

шаблоны для отображения данных и не требует контроллеров для обработки запросов.

4. Blazor — это структура проекта, которая позволяет создавать SPA-приложения с помощью C# и .NET. Она использует компонентную модель и может работать как на сервере, так и в браузере.

5. Minimal API — это структура проекта, которая позволяет писать минимум кода, отказываясь от многих паттернов и конструкций.

Особое место в данных архитектурах занимает именно MVC. На этой модели основаны многие примеры, она очень удобна для создания больших приложений и хорошо масштабируется. Структура MVC (Model-View-Controller) в ASP.NET Core представляет собой следующие элементы

1. Models — это классы, которые описывают данные и бизнес-логику приложения. Они могут содержать методы для работы с данными, валидацию и другие функции.

2. Views — это файлы, которые отображают данные пользователю. Они могут быть написаны на Razor-шаблонах или на других языках, таких как HTML, CSS и JavaScript.

3. Controllers — это классы, которые обрабатывают запросы пользователя и взаимодействуют с моделями и представлениями. Они могут содержать методы для получения данных из моделей, передачи данных в представления и другие функции.

4. Startup.cs — это файл конфигурации приложения, который содержит настройки сервисов, маршрутизации и другие параметры.

5. wwwroot — это папка, которая содержит статические файлы, такие как изображения, CSS и JavaScript.

Структура проекта MVC в ASP.NET Core может быть изменена в зависимости от требований приложения, но эти основные элементы остаются неизменными.

Рассмотрим структуру MVC на примере немного видоизмененного стартового проекта, который можно создать, выполнив команду dotnet new mvc.

При создании приложения будет создан ряд директорий, описанных выше, а конкретно

1. Controllers, содержащая контроллер HomeController для отображение домашней страницы и страницы безопасности

2. Models, содержащая модель данных ошибки, ErrorViewModel

3. Views, содержащая общие страницы разметки и представления для контроллера HomeController в соответствующей поддиректории Home (по названию контроллера)

4. wwwroot, содержащая минимальные стили для страниц

А так же файл Program.cs, в описана логика по созданию и запуску приложений. При создании приложения WebApplicationBuilder, с помощью него настраиваются сервисы, далее создается приложение, в котором настраивается конвеер обработки запроса, и затем приложение запускается

Листинг 2: Код файла Program.cs

```
// Созданиебилдера
var builder = WebApplication.CreateBuilder(args);

// Настройкасервисов
builder.Services.AddControllersWithViews();

// Созданиеприложения
var app = builder.Build();

// Настройкаконвеераобработкизапроса
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();

app.MapControllerRoute(
name: ''default'',
pattern: ''{controller=Home}/{action=Index}/{id?}'');

// Запускприложения
app.Run();
```

2.5. Сервер, исспользуемый в ASP.NET Core

По умолчанию, сервер, на котором запускается приложение — Kestrel[6]. Согласно документации Microsoft[6], Kestrel — это кроссплатформенный веб-сервер для ASP.NET Core. Kestrel — это рекомендуемый сервер для ASP.NET Core, и он настроен по умолчанию в шаблонах проектов ASP.NET Core. Преймущества Kestrel, согласно документации Microsoft:

1. Кроссплатформенность: Kestrel — кроссплатформенный веб-сервер,

работающий в Windows, Linux и macOS.

2. Высокая производительность: Kestrel оптимизирован для эффективной обработки большого количества одновременных подключений.

3. Простота использования: Kestrel оптимизирован для работы в средах с ограниченными ресурсами, например контейнерами и пограничными устройствами.

4. Безопасность: Kestrel поддерживает протокол HTTPS и защищен от уязвимостей веб-сервера.

5. Поддержка расширенных протоколов: Kestrel поддерживает общие веб-протоколы, в том числе: HTTP/1.1, HTTP/2 и HTTP/3, WebSockets

6. Гибкость рабочих нагрузок: Kestrel поддерживает множество рабочих нагрузок:

   (a) ASP.NET приложения, такие как Minimal API, MVC, Razor pages, BlazorSignalR и gRPC.

   (b) Создание обратного прокси-сервера с помощью YARP.

7. Расширяемость: настройка Kestrel с помощью конфигурации и ПО промежуточного слоя. Диагностика производительности: Kestrel предоставляет встроенные функции диагностики производительности, такие как ведение журнала и метрики.

Также, на платформе Windows, присутствуют HTTP.sys и IIS, а так же есть возможность написать свою реализацию сервера, воспользовавшись интерфейсом IServer.

2.6. Интеграционные тесты в ASP.NET Core

Интеграционное тестирование, в отличие от модульного тестирования, позволяет полностью протестировать работу системы. В случае

веб-приложения это означает, что тестируется все система, начиная от принятия запроса, заканчивая его обработкой и возвращением результата. Интеграционные тесты гарантируют, что компоненты приложения работают правильно на уровне, включающем поддерживающую инфраструктуру приложения, такую как база данных, файловая система и сеть. ASP.NET Core поддерживает интеграционные тесты с использованием платформы модульного тестирования с тестовым веб-узлом и тестовым сервером в памяти.

На ASP.NET интеграционное тестирование приложения делается с помощью WebApplicationFactory[7]. WebApplicationFactory предоставляет пользователю запустить приложение с сервером TestServer, которое будет работать в памяти. У этого сервера можно создавать клиентов, и моделировать взаимодействие с веб-приложением. Тесты, сделанные с помощью WebApplicationFactory, имеет схожую структуру.

1. Перед самим тестом, создается экземпляр WebApplicationFactory, в котором выполняется подстановка сервера, и запускается фактическое приложение в отдельном потоке, готовое обрабатывать пользовательские запросы. Так же, опционально, может проводиться дополнительная онфигурация, например мокирование некоторых объектов.

2. Сначала, в самом тесте, создается клиент. Он уже подключен к серверу, и любой запрос от него пойдет прямо в приложение.

3. Далее, с помощью метода, позволяющего совершать запросы, делается ряд запросов к серверу, при котором получаются данные, выполняются некоторые эффекты, например запись в базу или создание файла.

4. После, полученные ответы и эффекты проверяются на соответствие требованиям тестировщика

Листинг 3: Пример теста на простой API

```
private WebApplicationFactory<Program> _factory;

[SetUp]
public void Setup()
{
    // Созданиеобъекта  WebApplicationFactory
    _factory = new WebApplicationFactory<Program>();
}

[Test]
public async Task Test()
{
    // Созданиеклиента
    var client = _factory.CreateClient();

    // Взаимодействиессервером
    var response = await client.GetAsync("api/get");

    // Проверкарезультата
    response.EnsureSuccessStatusCode(); // Status
      Code 200-299
}
```

2.7. Существующие решения

Для тестирования веб-приложений на основе ASP.NET Core существует два основных подхода, фаззинг и модульное тестирование.

Фаззинг — это техника тестирования методом черного ящика, которая обычно состоит из автоматического нахождения багов реализации используя модифицированные/случайные наборы данных. Тестирование методом черного ящика означает, что при генерации тестовых данных программа не имеет доступа к коду тестируемого проекта. Это

значит, что могут быть не найдены редкие и труднодоступные уязвимости, которые можно найти, например, используя технику символьного исполнения.

В данный момент, инструменты, которые предоставляют символьное исследование кода веб-приложений, могут создавать только модульные тесты. При создании интеграционных тестов, то есть исследовании всего приложения, возникает большое количество символьных состояний, а также большое количество ситуаций, когда символьный движок встречает что-то, что не может исполнить.

Далее, мы рассмотрим следующие инструменты: RESTler и Pex.

2.7.1. RESTler

RESTler — это инструмент для фаззинга для автоматического тестирования облачных сервисов через их REST API и нахождения уязвимостей и ненадёжностей в этих сервисах, разработанный Microsoft в 2019 году[2].

Во время тестирования, RESTler динамически обучается на том, как ведет себя сервис в зависимости от предыдущих запросов. Собранные данные позволяет ему исследовать более глубокие состояния сервиса, доступные только при особых последовательностях запросов, что позволяет найти больше ошибок.

RESTler использует фаззинг, то есть тестирование методом черного ящика. Он не знает о том, что написано в коде, и это позволяет тестировать любые сервисы, которые поддерживают REST API, но при том критические, но маловероятные уязвимости могут быть пропущены. Также данный метод может занимать длительное время для нахождения уязвимостей.

RESTler является инструментом с открытым исходным кодом.

2.7.2. Pex

Pex — это инструмент для автоматического анализа программ на платформе .NET. Этот инструмент использует динамическое символь-

ное исполнение для нахождения всех возможных комбинаций входных данных. Pex доступен для разработчиков программ на платформе .NET, так как является частью среды разработки Visual Studio Enterprise под названием IntelliTest[4].

Pex использует тестирование методом белого ящика, то есть при генерации входных данных инструмент знает, как выглядит исходный код, и располагая данной информацией создает тесты с большим уровнем покрытия.

Pex можно использовать для генерации тестов с большим покрытием на отдельные модули, например на действия контроллеров, или проводить модульное тестирование, но интеграционные тесты на веб-приложения сделать не получится.

3. Архитектура

В данной главе описывается архитектура разработанной системы, а также представлены принципы, понятия и ключевые идеи, необходимые для проведения процесса исследования кода веб-приложений на основе ASP.NET Core, а также генерация и запуск полученных интеграционных тестов.

3.1. Исследование кода

Исследование кода веб-приложения на основе ASP.NET Core не представляется возможным, так как фактически приложение представляет собой не только настройку приложения и обработку запроса, но и запуск настоящего сервера, работу с HTTP и обработку сообщений сервером. Для исследования кода необходимо применить ряд модификаций к проекту V#, чтобы изменить логику исследования веб-приложений.

3.1.1. Алгоритм обработки запроса

В веб-приложениях на ASP.NET Core, мы хотим исследовать в основном ту часть, с которой будет взаимодействовать пользователь, то есть весь путь обработки запроса, от его получения, до выдачи ответа.

При обработке запроса в веб-приложениях на основе ASP.NET Core существует два основных типа компонентов — ПО промежуточного слоя и действия в контроллерах. ПО промежуточного слоя — это программное обеспечение, которое собирается в конвейер, который обрабатывает запросы и ответы. Каждый компонент:

- выбирает, передать ли запрос следующему компоненту в конвейере;
- может делать работу перед выполнением следующего компонента, и после нее.

Объекты класса `RequestDelegate` используются для построения конвейера обработки запроса, они обрабатывают каждый HTTP запрос[9].

Контроллер используется для определения и группировки набора действий. Действие (или метод действия) — это метод на контроллере, обрабатывающий запросы. Контроллеры логически группируют схожие действия[8]. Контроллер является классом, в котором по крайней мере верно одно из условий:

- имя класса имеет суффикс `Controller`;
- класс является наследником контроллера;
- к классу применяется аттрибут `Controller`.

Действия контроллера, являются стандартными методами и работают с объектами .NET, которые создаются на основе информации в HTTP запросе. Действие контроллера может принимать аргументы из разных частей запроса, из формы, заголовков, тела, пути или запроса. В ходе работы действие контроллера может изменять ответ, выставлять статус-код, проверять заголовки. Результат работы действия контроллера может быть записан в тело. Действия контроллеров — это конец конвейера, в них может быть реализована бизнес-логика, или обработка может быть делегирована соответствующим службам.

При обработке HTTP запроса, он сначала проходит по всем элементам ПО промежуточного слоя, потом обрабатывается внутри контроллера, а затем идет обратно по всей цепочке вызовов.

Конвейер обработки запроса изображен на Рис. 2.

3.1.2. Построение конвейера обработки приложения ASP.NET Core

В веб-приложении на основе ASP.NET Core, перед запуском сервера с конвейером обработки запроса, этот конвейер необходимо создать и настроить. Для создания приложения используются объект класса `WebApplicationBuilder`, в котором можно настроить сервисы, среду, конфигурацию, логирование и хост. Далее, приложение можно собрать, дополнительно настроить и запустить с помощью метода `Run`, который

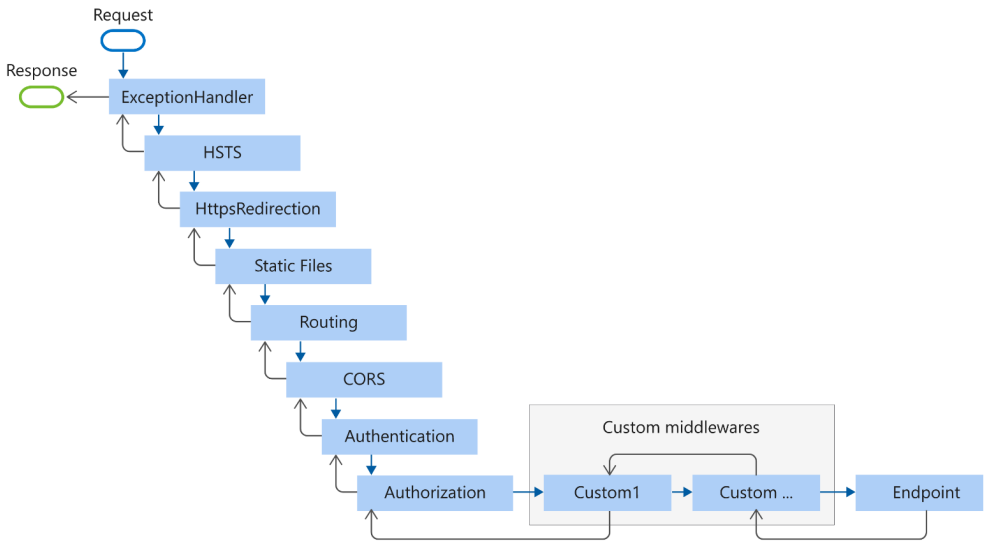

Рис. 2: Конвейер обработки запроса[9]

начинает работу сервера, с конвейером, созданным и настроенным на основе предыдущих вызовов.

3.1.3. Исследование конвейера обработки запроса

При исследовании работы веб-приложения необходимо исследовать только сконфигурированный конвейер обработки запроса, так как именно в нем находится весь алгоритм обработки запроса, на который необходимо создать интеграционные тесты.

Исследовать работу всего приложения сразу не получится, так как код, который написан пользователем, помимо настройки и создания интересующего нас конвейера, фактически начинает работу сервера, который в свою очередь ждет HTTP запросы по соответствующим адресам, и запускает обработку запросов, которые ему приходят.

Поэтому необходимо как-то прервать работу приложения в тот момент, когда конвейер создан и настроен, но сервер еще не начал работу, и к конвейеру можно получить доступ.

На Рис. 3 изображена схема символьного исследования приложения. Символьное исполнение начинается с точки входа, `Program.Main`. Далее, когда достигается конструктор `WebApplicationOptions`, с помощью модуля `Reflection` создается метод `ConfigureAspNet`. Он исполняется вместо конструктора, и в нем происходит переопределение

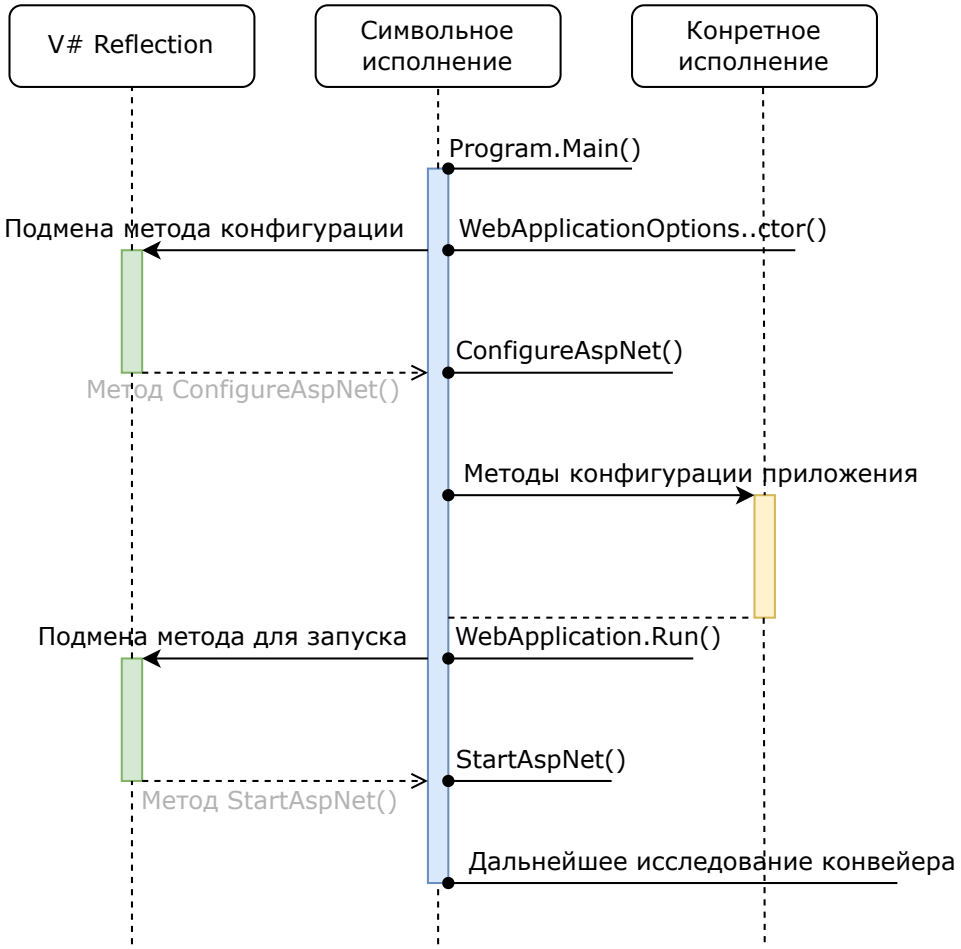

Рис. 3: Схема символьного исследования приложения

настроек приложения на конфигурацию, нужную для исследования. Далее, при исполнении точки входа, встречается много вызов, которые настраивают конвейер обработки запроса. Вызовы этих методов происходят конкретно. Далее, при запуске приложения, в методе `WebAppication.Run`, с помощью модуля `Reflection` создается метод `StartAspNet`. Созданный метод исполняется вместо запуска приложения, а также становится точкой входа. По завершении исполнения метода `StartAspNet`, получившиеся символьны состоянии используются для генерации интеграционных тестов.

Для реализации данного алгоритмы были созданы функции для генерации методов `StartAspNet`, `ConfigureAspNet` в модуле Reflection, изменен интерпретатор. Также были созданы функции для работы с Json, которые упрощают работу с сериализацией и десериализацией символьных объектов.

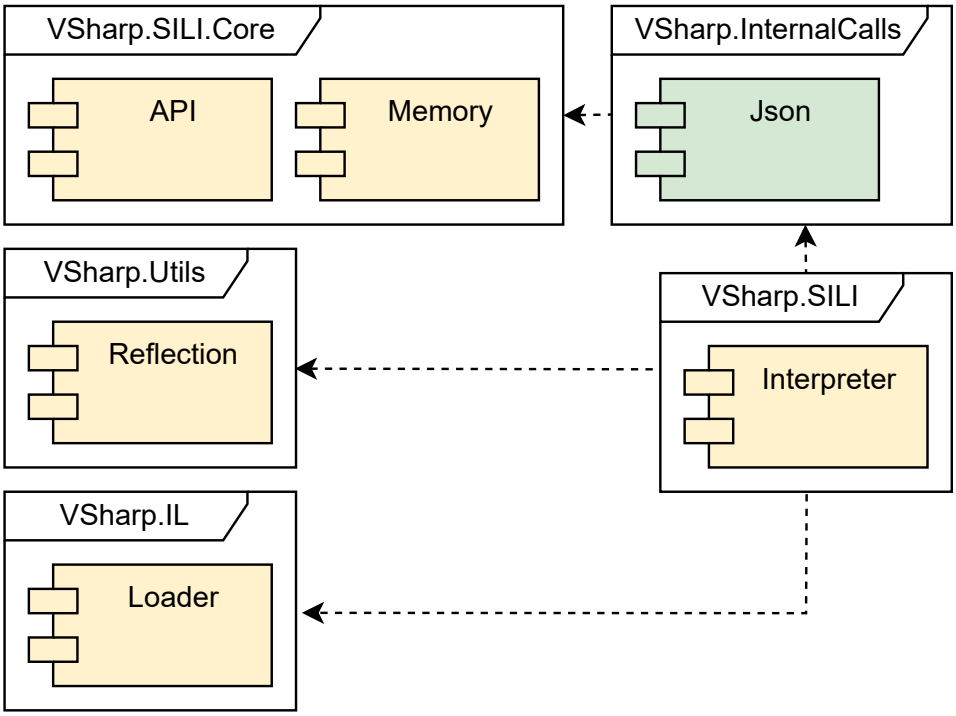

Рис. 4: Модули, модифицированные для работы с ASP.NET Core

3.1.4. Замена методов при исследовании

Для того чтобы прервать работу приложения в нужный момент, и выполнить необходимые действия для исследования конвейера, во время вызова определенных методов предлагается заменить их на созданные искусственные методы. Для этого создаются методы `StartAspNet` и `ConfigureAspNet`, и заменяются вызовы методов ASP.NET Core.

В методе `ConfigureAspNet`, который вызывается в момент создания настроек, конфигурируется объект типа `WebApplicationOptions`. В нем поля `EnvironmentName`, `ApplicationName`, `ContentRootPath` инициализируются согласно интересующий конфигурации.

В методе `StartAspNet`, который вызывается в тот момент, когда конвейер обработки запроса собран и настроен, перед запуском сервера, происходит запуск собранного объекта типа `RequestDelegate` с символьным запросом, и возвращение ответа, который был получен в качестве работы конвейера.

Методы создаются в модуле `Reflection`, с помощью объекта типа `ILGenerator`, который позволяет определить метод, используя его CIL

код. В модуле Interpreter, в котором осуществляется символьное исполнение пользовательского проекта, при вызове конструктора `WebApplicationOptions`, создается метод `ConfigureAspNet`, и вызывается вместо конструктора `WebApplicationOptions`. Далее, методы настройки приложения вызываются конкретно. В конце точки входа в приложение присутствует метод `WebApplication.Run`, который конфигурирует конвейер обработки запроса и запускает сервер. Внутри данного метода создается объект типа `HostingApplication`, и во время вызова конструктора `HostingApplication`, в модуле `Reflection` создается метод `StartAspNet`, и вызывается вместо конструктора `HostingApplication`. При этом он становится точкой входа, и после символьного исполнения метода `StartAspNet` символьные состояния будут готовы для их анализа для создания интеграционных тестов.

3.2. Создание и запуск тестов

Результатом работы является набор тестов, которые необходимо получать из символьных состояний в результате символьного исполнения. В главе 3.2.1 описывается представление интеграционного теста, а в главе 3.2.2 приводятся измененные компоненты, нужные для создания интеграционных тестов.

3.2.1. Структура и создание интеграционного теста

Интеграционный тест, который мы хотим получить, фактически является представлением HTTP запроса, а также ожидаемого HTTP ответа. Всю информацию можно, такие, как заголовки, путь, метод, запрос, тело можно хранить в строках.

Формат интеграционного теста отличается от модульного теста, который создает символьный движок V#, хотя имеется схожие методы.

Для создания интеграционного теста, нужно использовать информацию, которая имеется в символьном состоянии на момент завершения исполнения. С помощью SMT-решателя, аргументы становятся конкретными и сериализуются в строку с помощью объекта типа

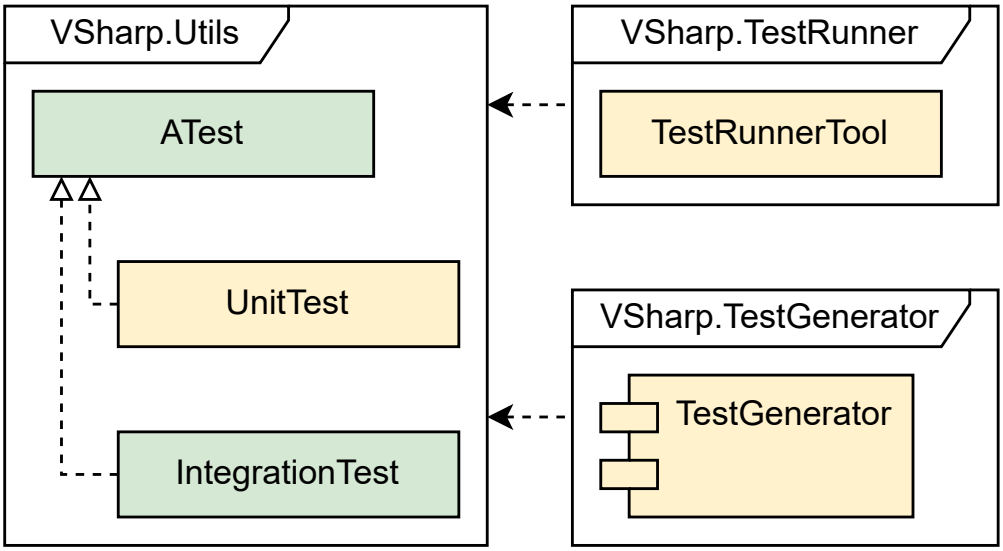

Рис. 5: Диаграмма классов для нового формата тестов

`System.Text.Json.JsonSerializer`. Ответ сервера также конкретизируется и сериализуется в строку. В соответствующие поля записываются поля запроса и ответа.

Для представления интеграционного теста был создан класс `AspIntegrationTest`, и абстрактный класс `ATest`, который имеет двух наследников, клас `UnitTest`, представляющий модульный тест, и новый класс `AspIntegrationTest`.

Был добавлен ряд новых функций для создания и запуска интеграционных тестов в модулях `TestGenerator` и `TestRunnerTool`. Подробно модификации `TestGenerator` будут разобраны в главе 4.4.2. Модификации генератора тестов, а алгоритм запуска тестов в `TestRunnerTool` будет описан в главе 4.4.3. Запуск интеграционных тестов.

3.2.2. Запуск интеграционных тестов

Чтобы запустить интеграционные тесты для веб-приложений, основанных на ASP.NET Core, которые были созданы в результате работы символьного движка, был использован класс `WebApplicationFactory`, который запускает приложение в памяти, позволяя делать к нему реальные запросы. Внутри `TestRunner` был создан метод, который инициализирует объект `WebApplicationFactory`, проводит необходимую настройку, посылает запрос и сверяет ответ с тем, что ожидается в тесте.

4. Реализация

В данной главе описывается реализация разработанной системы, работа с JSON, более подробная работа с методами `StartAspNet` и `ConfigureAspNet`, а также описывается работа с аргументами действий контроллера.

4.1. Работа с JSON

При символьном исследовании веб-приложений на основе фреймворка ASP.NET Core часто возникают ситуации, когда символьный объект надо сериализовать в строку, или наоборот, из символьной строки получить объект. Исследовать код методов сериализации и десериализации не представляется возможным из-за их сложности, и возможного роста количества символьных состояний.

Было принято решение использовать подход, который использует свойство сериализации и десериализации.

```
x = Deserialize(Serialize(x))
```

При вызове метода сериализации символьного объекта создается символьная константа, которая содержит внутри себя источник — ссылку на изначальный объект. При десериализации такой символьной константы результат извлекается из источника константы. Далее рассматриваются особенности следующего подхода.

1. Скорость работы — благодаря отсутствию нужды в символьном исполнении большого объема кода внутри методов сериализации и десериализации, существенно увеличивается скорость работы.

2. Легкость исполнения символьным движком — так как не исполняется код сериализации и десериализации, то это сокращает возможность появления проблем с внешними вызовами и повышает скорость работы интерпретатора.

3. Уменьшение количества символьных состояний — при работе с символьными строками или объектами могло бы получится боль-

шое число символьных состояний. Это очень сильно сказалось бы на общей производительности.

4. Универсальность — подобный подход работает с любыми объектами, как внутри кода ASP.NET Core, так и внутри пользовательского кода.

5. С результатом нельзя работать как со строкой — операции над сериализованным объектом как со строкой не поддержаны. Единственные варианты работы с информацией в формате Json это сериализация из объекта и десериализация в объект.

6. Неточность — символьная константа, полученная таким образом из объекта, будет представлять его полностью, в отличие от реальной сериализации, где свойства и поля объекта будут сохранены в соответствии с аттрибутами и видимостью.

7. Отсутствие работы с настройками сериализации — при сериализации объекта в Json, полученная строка строится по правилам, заданными объектом типа `JsonSerializerOptions`. Этот подход не учитывает эти настройки, в результате чего строка, полученная реальной сериализацией будет отличиться от строки полученной этим методом.

8. Дополнительные действия при создании тестов — при создании тестов не получится использовать строку напрямую, требуется проверять, что она является строкой, полученной в результате сериализации, и вызывать реальный метод сериализации на конкретизированном SMT-решателем объекте.

Проблема отсутствия работы с настройками сериализации была решена добавлением переданных в аргументах метода сериализации объекта настроек. Настройки сохраняются вместе со ссылкой на символьный объект, и при десериализации в тесте применяются. Таким образом методы, осуществляющие сериализацию и десериализацию теперь учитывают переданные настройки.

Также, важно упомянуть, что фактически, процесс сериализации и десериализации означают не передачу объекта как ссылки, а передачу его копии.

```
x = new CatPhoto()
y = Deserialize(Serialize(x))
z = Deserialize(Serialize(x))
```

То есть, если был создан и сериализован объект X, а Y и Z получились в результате сериализации, то они будут копиями объекта. Это важное свойство, так как аргументы, переданные в контроллер могут быть мутированны, что, учитывая предполагаемое ссылочное равенство двух объектов в некорректном представлении, может повлечь некорректный тест.

В ASP.NET Core работа с Json связана с объектом класса `Stream`, которые представляют собой универсальное представление последовательности байтов. Для обеспечения работы сериализации и десериализации, с помощью внутренних вызовов V# были переопределены методы `JsonSerializer.SerializeAsync` и `DeserializeAsync`, которые работают с объектами класса Stream.

Действия внутри переопределенного метода сериализации:

1. создается символьная константа типа байт;
2. в качестве источника константы выставляется переданный аргумент для сериализации;
3. в символьную константу сохраняются настройки сериализации;
4. в памяти выделяется массив байтов, в первый элемент кладется созданная константа;
5. массив байт записывается в поле `_buffer` переданного объекта типа `MemoryStream`;
6. возвращается `Task.CompletedTask`.

Действия внутри переопределенного метода десериализации:

1. у переданного объекта `MemoryStream` из поля `_buffer` берется массив байт;

2. из массива байт извлекается первый элемент, это символьная константа с нужным нам источником;

3. символьный объект достается из источника символьной константы;

4. создается объект типа `ValueTask<T>`, где T - тип указанный пользователем и переданный в метод;

5. тип, указанный пользователем, и тип символьного объекта сверяются, в случае несоответствия выбрасывается исключение;

6. символьный объект копируется;

7. в качестве результата созданного объекта типа `ValueTask<T>` выставляется копия символьного объекта;

8. возвращается объект типа `ValueTask<T>`.

4.2. Аргументы действия контроллера

Для достижения большого покрытия, все аргументы контроллера должны быть символьными, но для того, чтобы символьные аргументы, созданные при запуске метода `StartAspNet`, внутри запроса дошли до контроллера, нужно пройти огромное количество строк кода, в том числе сериализации и десериализации в различные форматы, чтобы из строк в HTTP запросе создать .NET объекты. Этот подход имеет ряд отрицательных свойств, особенно сказывается большое время работы.

Было принято решение создавать аргументы во время вызовов методов их десериализации из данных в HTTP контексте. При создании динамического метода `StartAspNet`, созданный метод имеет тип, который помимо ряда технических аргументов принимает все аргументы исследуемого контроллера, но внутри метода с ними ничего не делается. Перед вызовом метода все аргументы сохраняются в специальном

поле символьного состояния. Для того чтобы передать аргумент в контроллер, с помощью внутренних вызовов V# был переопределен метод `ParameterBinder.BindModelAsync`. Он используется для того, чтобы из данных, которые имеются в запросе, то есть заголовков, пути, запроса, тела, форм, собрать объект .NET заданного типа для известного аргумента действия контроллера на основе аттрибутов, которые определяют из какого источника брать информацию. На момент вызова метода `ParameterBinder.BindModelAsync` известно, какой конкретно аргумент мы хотим десериализовать. Вместо исполнения кода десериализации, аргумент, в соответствии со своей позицией, взятой из информации о параметре, берется из символьного состояния, копируется, и возвращается в качестве результата.

4.3. Замена методов при исследовании

Для организации процесса символьного исследования конвейера обработки запроса, необходимо заменять некоторые методы на методы, которые будут делать полезную для исследования работу.

4.3.1. Создание динамических методов

Common Intermediate Language (CIL) - набор инструкций для виртуальной системы исполнения. Именно в CIL транслируется код из высокоуровневых языков платформы .NET, и именно его символьно исполняет движок V#. В связи с тем фактом, что одинаковые типы, импортированные из разных сборок несовместимы, писать код внутри проекта V# в виде кода на C# или F# не представляется возможным.

Поэтому было принято решение использовать класс `ILGenerator`, который позволяет создать динамический метод, задать его сигнатуру, а также описать алгоритм метода с помощью инструкций CIL.

Для этого, в модуле `Reflection` были созданы функции `createAspNetStartMethod` и `createAspNetConfigureMethod`. Сначала, с помощью модуля `AssemblyManager`, из сборки пользователя извлекаются все необходимые для построения типы. Далее, задается сигна-

тура методов и с помощью `ILGenerator` прописываются инструкции методов. Эти функции вызываются в модуле Interpreter, в момент вызова методов, которые требуется заменить на искусственные. Далее, с помощью методов интерпретатора, создаются аргументы, и созданный метод вызывается с новыми параметрами.

4.3.2. Метод конфигурации

Метод конфигурации `ConfigureAspNet` используется для создания настроек приложения. Метод заменяет конструктор класса `WebApplicationOptions`. В методе происходит присвоение `EnvironmentName`, `ApplicationName`, `ContentRootPath` в переданный ему объект типа `WebApplicationOptions`.

4.3.3. Метод запуска

Метод запуска `StartAspNet` используется для запуска настроенного объекта конвейера обработки запроса. Метод заменяет конструктор `HostingApplication`, в который передается собранный объект типа `RequestDelegate`. Код метода `StartAspNet`, помимо запуска самого делегата, настраивает запрос в соответствии с переданными аргументами.

Далее перечислены аргументы метода `StartAspNet`.

1. Конвейер обработки запроса — на момент вызова, конвейер обработки запроса уже создан и настроен.

2. Фабрика для создания HTTP контекста — требуется для создания HTTP контекста.

3. Путь — путь до контроллера, который мы хотим исследовать.

4. Метод — метод запроса, например `GET`, `POST` или `DELETE`.

5. Дальнейшие аргументы — аргументы контроллера. Они дублируются для того, чтобы с ними удобно было работать при генерации тестов.

Действия алгоритма, описанные в коде метода `StartAspNet`:

1. создается объект класса `MemoryStream`, для хранения символьного тела запроса;
2. создается объект класса `FeatureCollection`, в котором будет храниться вся информация, связанная с запросом и ответом;
3. создается объект класса `HttpRequestFeature`, в котором хранятся детали запроса, такие как путь, метод, заголовки, тело;
4. инициализируются заголовки, в зависимости от аргументов ставится заголовок Content-Type, а также Host и Content-Length;
5. задается путь, метод запроса и созданный в начале `MemoryStream` используется в качестве тела запроса;
6. в `FeatureCollection` сохраняется `HttpRequestFeature`;
7. создается объект класса `HttpResponseFeature`, который хранит ответ;
8. уго поля инициализируются стандартными значениями;
9. в `FeatureCollection` сохраняется `HttpResponseFeature`;
10. с помощью переданной в конструктор фабрики HTTP контекста, создается стандартный HTTP контекст, который принимает в качестве аргумента конструктора настроенный объект класса `FeatureCollection`;
11. переданный в конструктор конвейер обработки запускается на созданном контексте, в результате чего мутирует ответ;
12. ответ из контекста читается и возвращается как результат работы функции.

При замене конструктора `HostingApplication` на метод `StartAspNet`, также меняется точка входа программы. Это сделано для того, чтобы

исследование закончилось в тот момент, когда нам пришел ответ, и на основе символьных состояний в результате можно было бы получить набор интеграционных тестов.

4.4. Генерация интеграционных тестов

Результатом символьного исполнения является набор символьных состояний. Далее описан процесс, в котором из символьного состояния создается интеграционный тест. В главе 4.4.1 подробно описан формат теста, а в главе 4.4.2 описаны модификации модулей и алгоритм запуска интеграционных тестов.

4.4.1. Формат интеграционных тестов

Интеграционный тест для веб-приложения на основе ASP.NET Core представляет собой HTTP запрос, а также HTTP ответ, который мы ожидаем получить, или исключение, которое возникнет в результате работы приложения. Также, необходимо иметь информацию о том, какое приложение нужно запускать, для этого нужно знать о пути до сборки с изучаемым проектом.

Для этого был создан класс `AspIntegrationTest`, в котором в тесте сохраняются следующие поля:

1. `assemblyPath` — путь до сборки с веб-приложением;
2. `requestPath` — путь для запроса;
3. `requestMethod` — метод запроса, такой как GET или POST;
4. `requestBody` — тело запроса или форма запроса;
5. `requestHeaders` — заголовки запроса;
6. `requestQuery` — запрос;
7. `responseBody` — тело ответа;
8. `responseStatusCode` — статус-код ответа.

Все описанные поля нужны для сохранения информации о запросе и ответе, а также достаточную информацию для запуска теста с помощью `WebAppicationFactory`

4.4.2. Модификации генератора тестов

Для создания интеграционных тестов был модифицирован модуль `TestGenerator`, который отвечает за создание теста из символьного состояния.

Были созданы функции `model2webTest` и `state2webTest`, которые используются для генерации интеграционных тестов. Внутри новых функций из символьного состояния создается интеграционный тест.

Аргументы, с которыми был запущен метод `StartAspNet`, новая точка входа, конкретизируются с использованием модели, и в соответствии с позицией в контроллере и аттрибутах у аргумента, заполняют соответствующие поля в тесте, такие как тело, заголовки, формы, запрос, путь.

Результат исполнения метода `StartAspNet` читается из результата символьного состояния, конкретизируется с использованием модели и формирует результат теста, статус-код ответа и тело ответа. Если результатом является исключение, это отражается в тесте.

Также была добавлена возможность работы с объектами класса `MemoryStream`, в которых была записана символьная константа с сериализованным в Json объектом. В случае, если в массиве имеется один элемент, и этот элемент является символьной константой содержащей сериализованный объект, объект извлекается из источника константы, конкретизируется с помощью модели, сериализуется в конкретную строку с использованием сохраненных настроек сериализации и записывается в массив байтов.

4.4.3. Запуск интеграционных тестов

Для запуска интеграционных тестов был изменен класс `TestRunnerTool`

Действия алгоритма запуска интеграционного теста

1. Загружается пользовательская сборка
2. Копируется файл с зависимостями, необходимы для работы `WebApplicationFactory`
3. С помощью рефлексии, создается объект класса `WebApplicationFactory`, с типовым параметром, типом из пользовательской сборки
4. С помощью рефлексии происходит настройка `WebApplicationFactory`, используется среда `''Production''`, отключается логирование.
5. Копируется файл `MvcTestingAppManifest.json`, он дополняется парой название сборки — путь до сборки, чтобы путь до пользовательской сборке был корректным при запуске приложения
6. С помощью `WebApplicationFactory` создается клиент, используя который можно отправить запрос на сервер и получить ответ
7. Настраивается запрос (путь, метод, тело, заголовки)
8. Клиент посылает запрос и возвращает результат

Для запуска данного алгоритма также был модифицирован метод сравнения объектов, который теперь более корректно работает с объектом класса `MemoryStream`, сравнивая их содержимое.

5. Тестирование

Для тестирования функциональности был создан ряд проектов, в которых есть контроллеры и ПО промежуточного слоя, где имеется арифметика, ветвление, аргументы контроллеров из разных источников, простые и комплексные типы данных. Также некоторые тесты намеренно содержат ситуации, которые не поддерживает полученная система для генерации тестов для веб-приложений.

5.1. Тестируемые проекты

Для тестирования было создано три небольших проекта. В первом проекте находится один контроллер, с несколькими действиями которые могут принимать в качестве аргументов как числа, так и более комплексные данные. В качестве комплексных данных было принято решение взять класс `Wallet`, в котором имеется свойство `MoneyAmount` типа `int`. Всего было написано четыре действия контроллера, код методов можно найти на Листинге 4.

Первое действие, `GetSum`, доступен по адресу `/simple/sum` и методу `GET`. При этом запросе, читается одно число из тела запроса, одно число из заголовка, и выдается их сумма. Данный метод является наиболее примитивным, он служит для тестирования того, что программа может создать и настроить конвейер обработки запроса, пройти все ПО промежуточного слоя, дойти до действия контроллера и корректно отработать.

Второе действие, `GetSumPositive`, доступен по адресу `/simple/sum_positive` и методу `GET`. По алгоритму работы сильно схож с первым действием, `GetSum`, но при этом имеет возможность выбросить исключение, если аргументы, переданные ему, меньше нуля. Данный метод позволяет протестировать умение системы работать с исключениями, произошелдшими во время работы программы, а также с ветвлениями.

Третье действие, `Complex`, доступен по адресу `/simple/complex` и методу `POST`. Этот метод работает уже не только с числами, но и с

комплексными типами, которые могут быть переданны в контроллер. Внутри он содержит много логики, математических операций и ветвления, а также принимает аргументы в контроллер из всех возможных источников HTTP запроса. Также метод имеет не только возвращаемое значение, но и тенденцию мутировать статус-код ответа. Это действие является самым сложным, он тестирует работу с ветвлениями, со сложными типами в контроллерах, и с модификациями статус-кода.

Четвертое действие содержит неподдержанную операцию — работу с базой данных Sqlite и взаимодействие с EntityFramework. Это действие позволяет продемонстрировать функциональность, которая пока не поддержана.

Во втором проекте помимо контроллера, содержащего первое и второе действие, есть еще ПО промежуточного слоя, которое модифицирует результат работы, заменяя текст ответа на строку `''Hello from middleware!''`, если после контроллера статус-код был равен 200. Всего было написано два действия контроллера, и одно ПО промежуточного соля. Код методов можно найти на Листинге 5, а код ПО промежуточного слоя в Листинге 6.

Методы контроллеров похожи на методы в первом проекте, но во втором проекте они служат для тестировании работы ПО промежуточного слоя. На данный момент, из-за большого количества достаточно сильных оптимизаций, которые не учитывали ПО промежуточного слоя, из него не получится мутировать запрос, и увидеть, что это отразится в контроллере. Но можно мутировать результат запроса, или выполнять действия, не связанные с ним, например логирование.

В третьем проекте есть ПО промежуточного слоя, которое пытается изменить аргументы у контроллера.

5.2. Характеристики тестового стенда

Тестирование реализованной функциональности проводилось на тестовом стенде с характеристиками, указанными в таблице 1

Таблица 1: Характеристики тестового стенда

ОС	Windows 10 Home Edition
CPU	AMD Ryzen 5 5600X 6-Core Processor 3.70 GHz
RAM	32 Gb DDR4
Версия .NET	7.0.404

5.3. Результаты

В покрытие было включено ПО промежуточного слоя и метод контроллера, который мы изучали. Измерялось время работы системы по генерации интеграционных тестов для веб-приложений, итоговое покрытие и количество сгенерированных тестов. Результаты измерений отражены в таблице 2.

Таблица 2: Результаты тестирования

Путь	Время	Покрытие	Кол-во тестов
`/simple/sum`	00:00:50.15	100%	1
`/simple/sum_positive`	00:01:10.75	100%	3
`/simple/complex`	00:11:01.75	100%	24
`/simple/get_wallets`	00:00:34.86	0%	0
`/middleware1/sum_ok`	00:01:02.61	52%	1
`/middleware1/sum_p_sco`	00:02:08.50	55%	3
`/middleware2/sum`	00:02:08.50	43%	1

Три действия контроллера из первого проекта полностью покрыты тестами, притом все тесты являются корректными. Это свидетельствует о том, что символьное исполнение контроллера работает с аргументами из любых источников, запроса, пути, заголовков, формы и тела. На действие `GetWallets` не было сгенерированно тестов, из-за отсутствия поддержки баз данных и фреймворка EntityFramework. Во втором проекте, был создан один тест `GetSumOk` с покрытием 52%. Это связано с тем, что одна из веток в коде ПО промежуточного слоя является семантически недостижимой. Тесты, полученные для `GetSumPositiveStatusCodeOverride` имеют большее покрытие, так как покрывают все ветки ПО промежуточного слоя. При генерации тестов для третьего проекта, из-за невозможности работы с заголовками в ПО

промежуточного слоя из HTTP контекста напрямую, некоторые ветки исполнения не удалось покрыть. Поддержку чтения заголовков планируется реализовать в процессе дальнейшей работы.

Исходный код проектов, которые были использованы при тестировании, можно найти в репозитории, расположенном по адресу
https://github.com/arthur100500/AspNetApps

Заключение

В ходе работы были получены следующие результаты:

1. выполнен обзор существующих решений, а именно: RESTler и Pex;
2. создан алгоритм для генерации интеграционных тестов для веб-приложений на основе ASP.NET Core, который предполагает замену методов в процессе исследования при конфигурации и запуске конвейера обработки запроса;
3. спроектирована архитектура решения, которая состоит из исполнения конфигурации веб-приложения, символьного исследования кода конвейера обработки запроса и генерации интеграционных тестов;
4. спроектированное решение реализовано на языках C# и F# в проекте V#;
5. проведено тестирование на созданных веб-приложениях на основе ASP.NET Core.

Список литературы

[1] Microsoft. — .NET. — URL: https://dotnet.microsoft.com/en-us/ (дата обращения: 2023-12-23).

[2] Microsoft. — RESTler. — URL: https://github.com/microsoft/restler-fuzzer (дата обращения: 2024-02-06).

[3] StackOverflow. — Stack overflow developer survey. — URL: https://survey.stackoverflow.co/2023/ (дата обращения: 2023-12-23).

[4] Tillmann Nikolai, de Halleux Jonathan. — Pex. — Microsoft. — URL: https://link.springer.com/chapter/10.1007/978-3-540-79124-9_10 (дата обращения: 2024-02-06).

[5] Microsoft. — Документация ASP.NET Core. — URL: https://learn.microsoft.com/ru-ru/aspnet/core/?view=aspnetcore-8.0 (дата обращения: 2023-12-23).

[6] Microsoft. — Документация Microsoft. Сервер Kestrel. — URL: https://learn.microsoft.com/ru-ru/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-8.0 (дата обращения: 2023-12-23).

[7] Microsoft. — Интеграционное тистирование ASP.NET Core. — URL: https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-8.0 (дата обращения: 2023-12-23).

[8] Microsoft. — Обработка запросов с помощью контроллеров в ASP.NET Core. — URL: https://learn.microsoft.com/ru-ru/aspnet/core/mvc/controllers/actions?view=aspnetcore-8.0 (дата обращения: 2024-01-06).

[9] Microsoft. — ПО Промежуточного слоя ASP.NET Core. — URL: https://learn.microsoft.com/en-us/aspnet/core/

fundamentals/middleware/?view=aspnetcore-8.0 (дата обращения: 2024-01-06).

[10] VSharp-team. — Репозиторий V#. — URL: https://github.com/VSharp-team/VSharp (дата обращения: 2023-12-23).