

[КАК СТАТЬ АВТОРОМ](#)

Сезон Big Data: можем, умеем, практикуем

**894.41**

Рейтинг

**OTUS**

Цифровые навыки от ведущих экспертов

**MaxRokatansky**

24 ноя 2020 в 10:34

## Тестируем веб-API ASP.NET Core

**8 мин****18K**

Блог компании OTUS, .NET\*, ASP\*, API\*, Тестирование веб-сервисов\*

[Перевод](#)

Автор оригинала: Chris Woodruff

Привет, Хабровчане! Для будущих учащихся на курсе "C# ASP.NET Core разработчик" публикуем перевод полезной статьи.





При проектировании и разработке широкого спектра API с использованием ASP.NET Core 2.1 Web API важно понимать, что это только первый этап в создании продуктивного и стабильного решения. Наличие стабильной среды для вашего решения также очень важно. Ключ к отличному решению заключается не только в правильном построении API, но и в его тщательном тестировании, чтобы исключить возможность негативного опыта у пользователей во время использования вашего API.

Эта статья является продолжением моей предыдущей статьи для InfoQ под названием «[Продвинутая архитектура веб-API ASP.NET Core](#)». Не беспокойтесь, вам не нужно вникать в предыдущую статью, чтобы разобраться с тестированием в этой, но она может помочь вам лучше понять, как я спроектировал обсуждаемое здесь решение. На протяжении последних нескольких лет я много времени размышлял о тестировании, создавая API для клиентов. Знание архитектуры веб-API ASP.NET Core 2.1 может помочь и вам расширить ваше понимание.

Сююшн и весь код из примеров в этой статье можно найти в [моем GitHub репозитории](#).

## Букварь веб-API ASP.NET Core

Давайте вкратце рассмотрим .NET и ASP.NET Core. ASP.NET Core — это новый веб-фреймворк, созданный корпорацией Майкрософт в качестве замещающей альтернативы устаревшей технологии, существующей со времен ASP.NET 1.0. Отказавшись от устаревших зависимостей и разработанный с нуля, фреймворк ASP.NET Core 2.1 спроектирован для кросс-платформенного выполнения и дает разработчику гораздо большую производительность.

## Что такое модульное тестирование?

Для некоторых людей тестирование программного обеспечения может быть в новинку, но ничего сложного в нем нет. Начнем с модульного тестирования (или юнит-тестирования). Формальное определение из Википедии — это «процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.» Я предпочитаю использовать более доступное определение: модульное тестирование используется для того, чтобы убедиться, что после добавления нового функционала или исправления багов код в вашем приложении работает должным образом. Мы тестируем небольшой фрагмент кода, чтобы убедиться, что он соответствуем нашим ожиданиям. Давайте посмотрим на образец модульного теста:

```
[Fact]
public async Task AlbumGetAllAsync()
{
    // Arrange

    // Act
    var albums = await _repo.GetAllAsync();

    // Assert
```

```
Assert.Single(albums);  
}
```

Хороший модульный тест состоит из трех частей. Первая — это **Arrange** часть, которая используется для подготовки любых ресурсов, которые могут понадобиться вашему тесту. В приведенном выше примере мне не требуется никакая подготовительная настройка, поэтому часть **Arrange** пуста (но я все еще оставляю комментарий к ней). Следующая часть, называемая **Act**, — это часть, в которой выполняется тестируемое действие. В этом примере я вызываю репозиторий данных для сущности типа **Album**, чтобы получить весь набор альбомов из источника данных, который использует репозиторий. В последней части теста мы убеждаемся или утверждаем (**Assert**) что результат выполненного действия был правильным. В этом тесте я проверяю, что получил только один альбом из репозитория данных.

Я буду использовать в этой статье инструмент для модульного тестирования xUnit. xUnit — это пакет с открытым исходным кодом для .NET Framework и .NET Core. Мы рассмотрим версию xUnit для .NET Core, которая входит в комплект поставки .NET Core 2.1 SDK. Вы можете создать новый Unit Test проект с помощью команды `cli .NET Core dotnet test`, либо из шаблона xUnit Test проекта в вашей любимой IDE, такой как Visual Studio 2017, Visual Studio Code или JetBrains Rider.

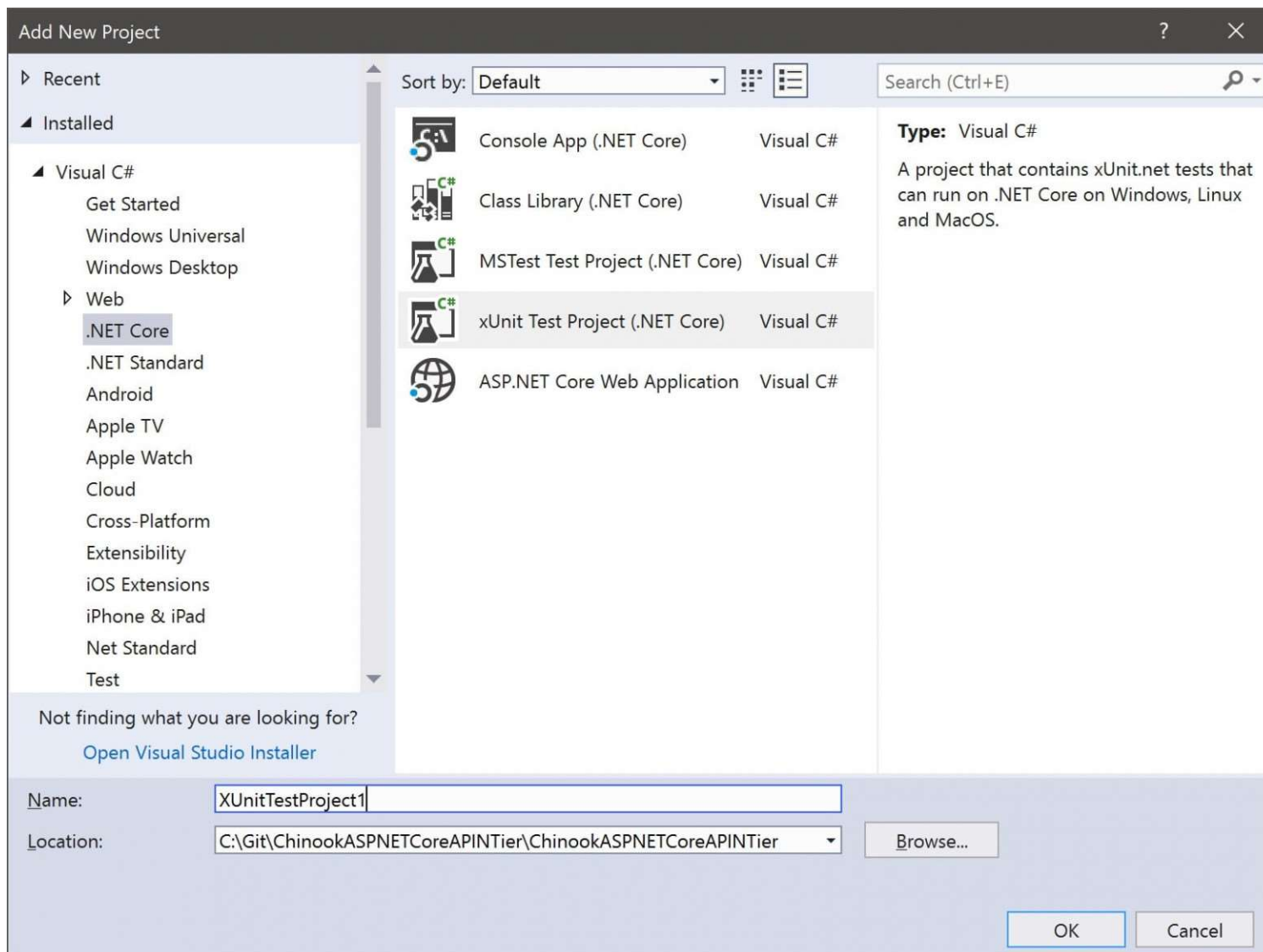


Рисунок 1: Создание нового Unit Test проекта в Visual Studio 2017

Теперь давайте перейдем к модульному тестированию вашего веб-API ASP.NET Core.

## Как следует тестировать веб-API?

Я большой сторонник использования модульного тестирования для поддержания стабильности и надежности API для ваших пользователей. Но я подхожу с умом к тому, как я использую свои модульные тесты и что тестирую. Моя философия заключается в том, что вы должны реализовать модульное тестирование своего проекта ровно настолько, насколько это необходимо. Что я имею в виду? Я могу получить много гневных комментариев за эту точку зрения, но меня не слишком заботит 100%-ное покрытие тестами. Считаю ли я, что нам нужны тесты, которые охватывают важные части API и изолируют каждую область независимо, чтобы гарантировать, что контракт каждого сегмента кода соблюден? Конечно! Это именно то, как я делаю и что хочу обсудить.

Поскольку наш демо проект Chinoook.API очень легковесный, и для него можно провести интеграционное тестирование (об этом чуть позже), я обнаружил, что больше всего концентрируюсь на модульных тестах в моих Domain и Data проектах. Я не буду вдаваться в подробности о том, как вам следует проводить модульное тестирование (поскольку эта тема выходит за рамки этой статьи). Я хочу, чтобы вы протестировали как можно больше ваших Domain и Data проектов, используя данные, которые не зависят от вашей производственной базы данных. Это следующая тема, которую мы рассмотрим, под названием «Моки данных и объектов».

### **Зачем использовать моки данных/объектов в ваших модульных тестах?**

Мы рассмотрели, что и зачем нам нужно для модульного тестирования. Важно также понимать, как правильно выполнить модульное тестирование кода веб-API ASP.NET Core. Данные являются ключом к тестированию вашего API. Вам необходимо иметь предсказуемый набор данных, который вы можете протестировать. Вот почему я бы не рекомендовал использовать производственные данные или любые данные, которые могут изменяться со временем без вашего ведома. Нам нужен стабильный набор данных, чтобы убедиться, что все модульные тесты выполняются и подтверждают выполнение контракта между сегментом кода и тестом. В качестве примера, когда я тестирую проект Chinoook.Domain для получения альбома (Album) с ID 42, я хочу быть уверен, что он существует и имеет ожидаемые от него детали, такие как имя альбома, и он связан с исполнителем (Artist). Я также хочу быть уверенным, что когда я получаю набор альбомов из источника данных, я получаю ожидаемую форму и размер, которые соответствуют написанному мной модульному тесту.

Многие коллеги используют термин «моки» (mocks или заглушки) для обозначения этого типа данных. Есть много способов сгенерировать моки данных для модульных тестов, и я надеюсь, что вы создадите как можно более «реальный» набор данных. Чем лучше ваши данные, которые вы создадите для своих тестов, тем лучше будет ваш тест. Я бы посоветовал вам в первую очередь убедиться, что ваши данные свободны от проблем с приватностью и не содержат личных или конфиденциальных данных вашей компании или вашего клиента.

Чтобы удовлетворить нашу потребность в чистых, стабильных данных, я создаю уникальные проекты, которые инкапсулируют моки данных для моих Unit Test проектов. В целях наглядности назовем мой проект с моками `Chinook.MockData` (как вы можете видеть в исходном демонстрационном коде). Мой `MockData` проект почти идентичен моему обычному проекту `Chinook.Data`. Он имеет одинаковое количество репозитория данных, и каждый из них реализует одни и те же интерфейсы. Я хочу, чтобы `MockData` проект хранился в контейнере внедрения зависимостей (Dependency Injection — DI), чтобы проект `Chinook.Domain` мог использовать его так же, как проект `Chinook.Data`, подключенный к источнику производственных данных. Вот за что я люблю внедрение зависимостей. Это позволяет мне переключать `Data` проекты в конфигурации без каких-либо изменений в коде.

## Интеграционное тестирование: а это еще что за вид тестирования веб-API?

После того, как мы выполнили и проверили модульные тесты для нашего веб-API ASP.NET Core, мы рассмотрим другой тип тестирования. Я использую модульное тестирование, чтобы проверить и подтвердить ожидания в отношении внутренних компонентов решения. Когда мы удовлетворены качеством внутренних тестов, мы можем перейти к тестированию API из внешнего интерфейса, что и называется интеграционным тестированием.

Интеграционные тесты следует писать и выполнять после завершения работы над всеми компонентами, чтобы ваше API могло быть использовано с правильным HTTP-ответом для проверки. Я смотрю на модульные тесты как на тестирование независимых и изолированных

сегментов кода, в то время как интеграционные тесты используются для тестирования всей логики каждого API на моем HTTP эндпоинте. Это тестирование будет исследовать весь рабочий процесс API от контроллеров API проектов до супервизоров Domain проектов и, наконец, репозитория Data проектов (и весь путь обратно до ответа).

## Создание проекта для интеграционного тестирования

Приобретенные на данный момент знания о тестировании пригодятся и здесь - мы реализуем функционал интеграционного тестирования на основе уже имеющихся у нас библиотек модульного тестирования. Я буду использовать xUnit для создания своих интеграционных тестов. После того, как мы создали новый тестовый проект xUnit с именем *Chinook.IntegrationTest*, нам нужно будет добавить соответствующий пакет NuGet. Добавьте пакет `Microsoft.AspNetCore.TestHost` в проект *Chinook.IntegrationTest*. Этот пакет содержит ресурсы, необходимые для выполнения интеграционного тестирования.





Рисунок 2: Добавление пакета NuGet Microsoft.AspNetCore.TestHost

Теперь мы можем перейти к созданию нашего первого интеграционного теста для внешней проверки нашего API.

## Создание вашего первого интеграционного теста

Чтобы начать внешнее тестирование всех API в нашем солюшене, я собираюсь создать новую папку под названием API, в которой будут храниться наши тесты. Я также создам новый тестовый класс для каждого из типов сущностей (Entity) в нашем домене API. Наш первый интеграционный тест будет покрывать тип сущности «Album».

Создайте в папке API новый класс с именем *AlbumAPITest.cs*. Теперь мы добавим в наш файл следующие пространства имен.

```
using Xunit;  
using Chinook.API;  
using Microsoft.AspNetCore.TestHost;  
using Microsoft.AspNetCore.Hosting;
```



Рисунок 3: Интеграционный тест с использованием директив

Для выполнения тестов нам нужно настроить в нашем классе `TestServer` и `HttpClient`. Нам нужна приватная переменная с именем `_client` типа `HttpClient`, которая будет создана на основе `TestServer`, инициализированного в конструкторе класса `AlbumAPITest`. `TestServer` — это обертка для небольшого веб-сервера, созданного на основе класса `Startup` `Chinook.API` и желаемой среды разработки. В этом случае я использую среду разработки `Development`. Теперь у нас есть веб-сервер, на котором работает наш API, и клиент, который понимает, как вызывать API в `TestServer`. Теперь мы можем писать код для интеграционных тестов.

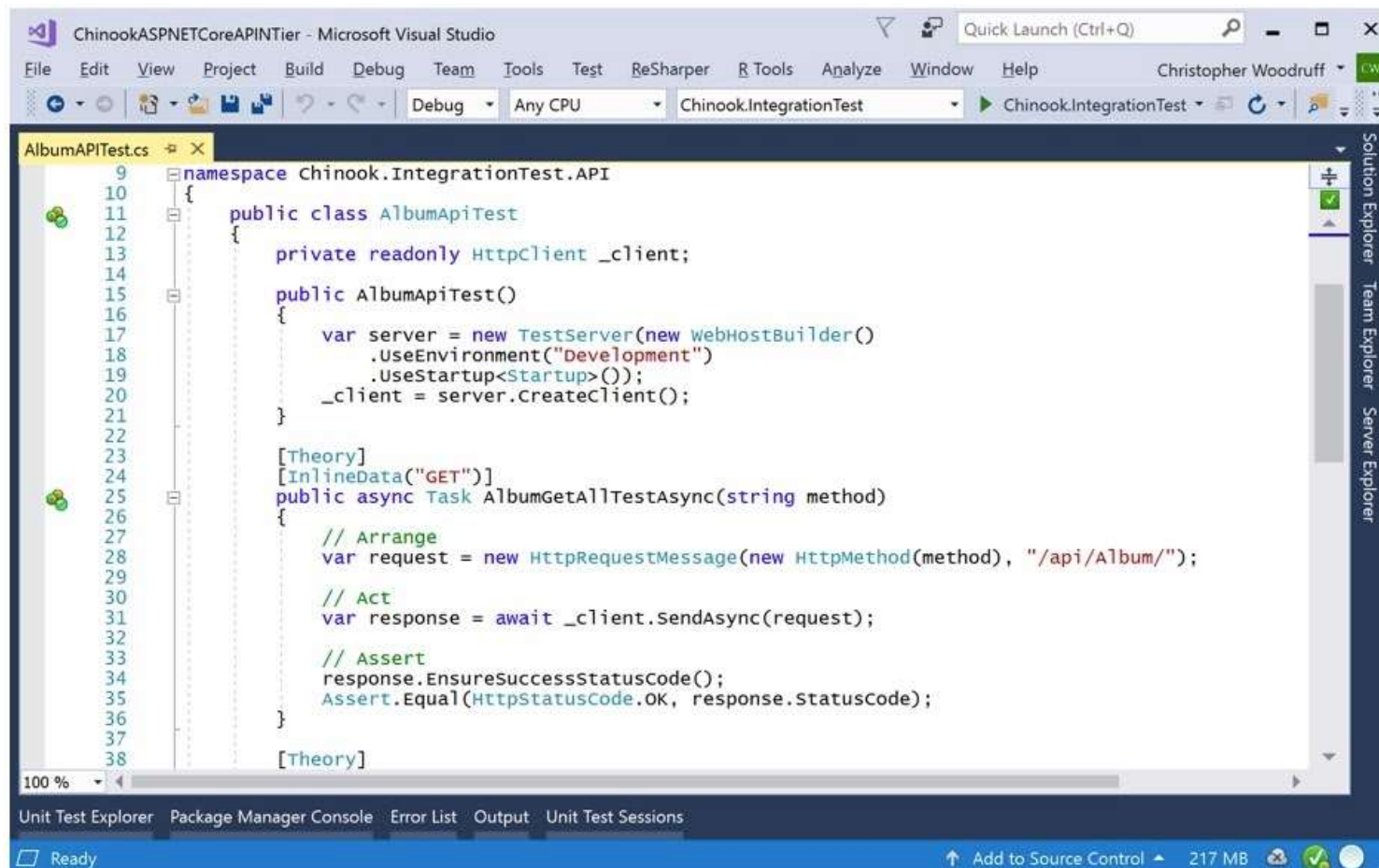


Рисунок 4: Наш первый интеграционный тест для получения всех альбомов

Помимо кода конструктора, на рисунке 4 также показан код для нашего первого интеграционного теста. Метод `AlbumGetAllTestAsync` проверит, работает ли вызов из API для получения всех альбомов. Как и в предыдущем разделе, где мы обсуждали модульное тестирование, логика нашего интеграционного тестирования также использует паттерн `Arrange/Act/Assert`. Сначала мы создаем объект `HttpRequestMessage` с HTTP-командой, предоставляемой в виде переменной из аннотации `InlineData`, и сегментом URI, который представляет вызов для запроса всех альбомов («/api/Album/»). Затем мы попросим `_client` `HttpClient` отправить HTTP-запрос, и, наконец, мы проверим, соответствует ли HTTP-ответ нашим ожиданиям, которые в данном случае — 200 OK.

На рисунке 4 я показал два способа проверить наш вызов API. Вы можете использовать любой из них, но я предпочитаю второй способ, поскольку он позволяет мне использовать тот же шаблон для проверки ответов для определенных кодов HTTP-ответов.

```
response.EnsureSuccessStatusCode();  
Assert.Equal(HttpStatusCode.OK, response.StatusCode);
```

Мы также можем создавать интеграционные тесты, которые должны проверять определенные ключи сущностей из нашего API. Для этого типа тестов нам нужно добавить дополнительное значение в аннотацию `InlineData`, которая будет передаваться через параметры метода `AlbumGetTestAsync`. Наш новый тест следует той же логике и использует те же ресурсы, что и предыдущий, но мы еще должны передать ключ сущности в сегменте URI для объекта `HttpRequestMessage`. Код вы можете увидеть на рисунке 5.



Рисунок 5: Второй интеграционный тест для сущности Album

После того, как вы написали все интеграционные тесты, вам нужно будет запустить их через Test Runner и убедиться, что все они пройдены. Все созданные вами тесты также могут быть выполнены в рамках DevOps во время процесса непрерывной интеграции (Continuous Integration - CI) — для автоматического тестирования вашего API в течение всего процесса разработки и развертывания. Что ж, теперь у вас есть представление о том, что нужно делать, чтобы ваш API хорошо проверялся и поддерживался на этапах разработки, проверки качества и развертывания, чтобы пользователи API могли работать без ненужных происшествий.

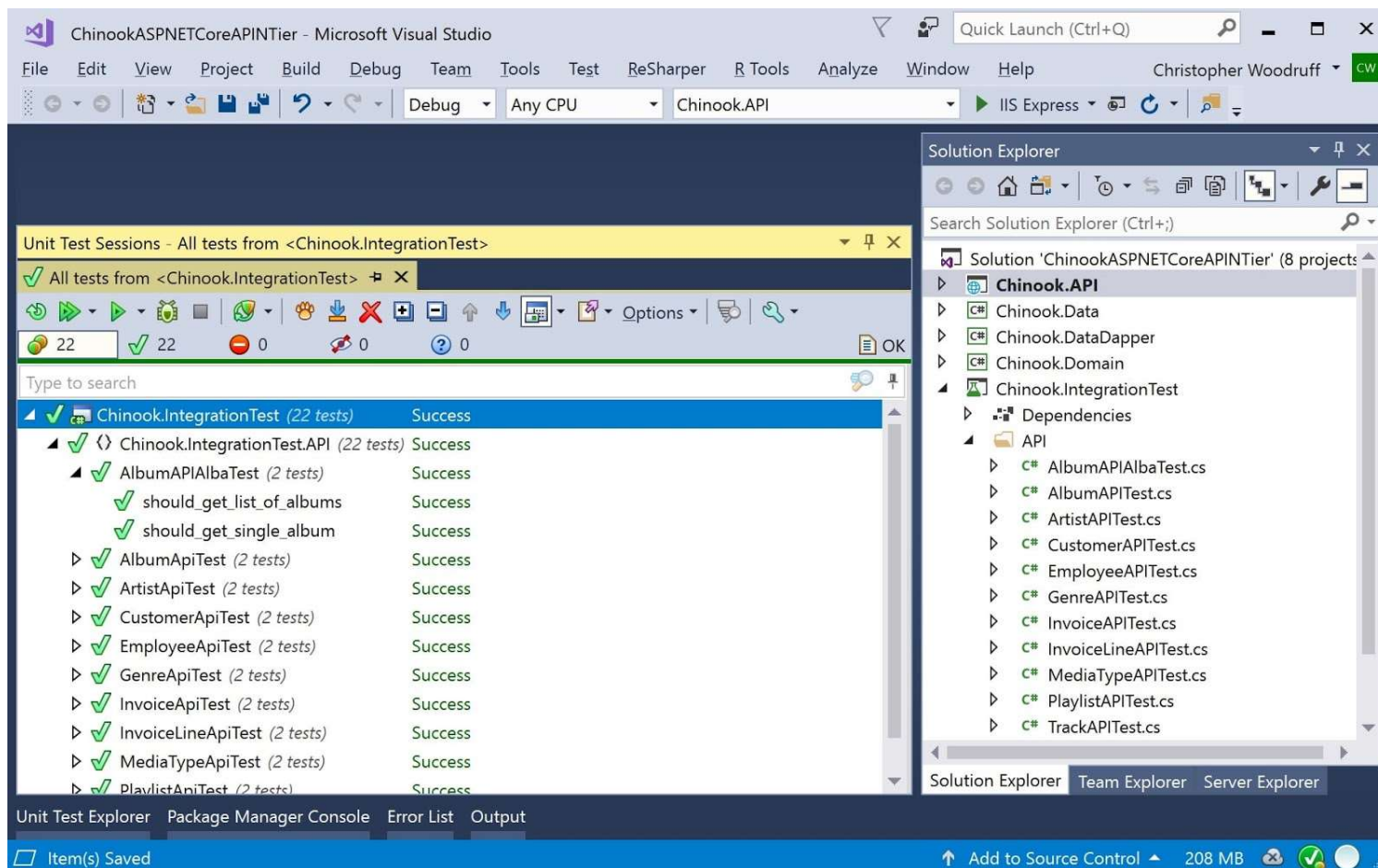


Рисунок 6: Выполнение интеграционных тестов в Visual Studio 2017

## Заключение

Наличие хорошо продуманного плана тестирования, использующего как модульное тестирование для проверки внутренних компонентов, так и интеграционное тестирование для проверки внешних вызовов API, так же важно, как и архитектура вашего веб-API ASP.NET Core.

Узнать подробнее о курсе "C# ASP.NET Core разработчик". Посмотреть запись открытого урока "Разработка GraphQL API на ASP.NET Core" можно [здесь](#).

**Теги:** asp.net core, тестирование web-api, .NET, модульное тестирование, интеграционные тесты

**Хабы:** Блог компании OTUS, .NET, ASP, API, Тестирование веб-сервисов

## Редакторский дайджест



Присылаем лучшие статьи раз в месяц

Электронная почта



OTUS

Цифровые навыки от ведущих экспертов

[Сайт](#) [ВКонтакте](#) [Telegram](#)



69

133.2

Карма Рейтинг

**OTUS** @MaxRokatansky

Редактор



Комментарии 11

## Публикации



ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ



Writer

18 часов назад

## Picodata: простое масштабирование Tarantool



Средний



16 мин



2.3K

Обзор



+68



26



1



kobzev\_кнопка

18 часов назад

## Про российский GitHub



4 мин



19K

Мнение



+47



28



138



SLY\_G

вчера в 11:12

## Британская паровая империя



12 мин



4.2K

Перевод



+36



20



2

**Bluewolf**

13 часов назад

## А что, если сделать Еще Один НеФлиппер?

**Простой**

2 мин



8.8K

Мнение

**+34**

29



43

**ru\_vds**

19 часов назад

## Реализуем с нуля функцию косинуса на языке C

**Сложный**

10 мин



4.3K

Тutorial

Перевод

**+32**

32



21

Показать еще

### ВАКАНСИИ КОМПАНИИ «OTUS»

Product Manager EdTech

от 80 000 Р · OTUS · Можно удаленно



Больше вакансий на Хабр Карьере



### ИНФОРМАЦИЯ



Сайт	otus.ru
Дата регистрации	22 марта 2017
Дата основания	1 апреля 2017
Численность	101–200 человек
Местоположение	Россия
Представитель	OTUS


## НОВОСТИ


---


 Моя прежняя работа стала как чемодан без ручки...  Мария Радосавлевич 15 лет пр...  
вчера в 18:01



 #Otus\_анонс  Анонс вебинаров в OTUS! Записывайтесь на вебинар и подключайтесь к тр...  
вчера в 11:01

Разыскиваются новые члены команды OTUS!  Product manager  Интернет-маркетолог ...  
18 июня

 Новая подборка самых интересных вебинаров недели в OTUS, которые помогут прокача...  
18 июня

  
17 июня

 Какой ОС вы пользуетесь?  
16 июня

 Задумываетесь о смене профессии?  В июле OTUS подготовил для вас запуск новых к...  
16 июня

 Уже в 19:00 мск состоится онлайн-митап «Как реагировать на факапы сотрудников?» Н...

15 июня

 Unity – это игровой движок. Платформа, которая позволяет писать уникальные програ...

15 июня

 Audio

14 июня

## ВИДЖЕТ



## В КОНТАКТЕ

БЛОГ НА ХАБРЕ

---

16 часов назад

**Система сохранения на Unity для начинающих** 1K  3

17 часов назад

**Трассировка распределенных IoT приложений на EMQX** 359  0

19 часов назад

**Изменение бизнес-моделей ведения бизнеса с использованием цифровых технологий** 357  0

20 часов назад

**Важные качества Team Lead** 1.1K  1

16 июн в 17:01

**Трюки со временем в 1C** 2K  10

Ваш аккаунт

Войти  
Регистрация

Разделы

Статьи  
Новости  
Хабы  
Компании  
Авторы  
Песочница

Информация

Устройство сайта  
Для авторов  
Для компаний  
Документы  
Соглашение  
Конфиденциальность

Услуги

Корпоративный блог  
Медийная реклама  
Нативные проекты  
Образовательные программы  
Стартапам  
Спецпроекты



Настройка языка

Техническая поддержка

Вернуться на старую версию

