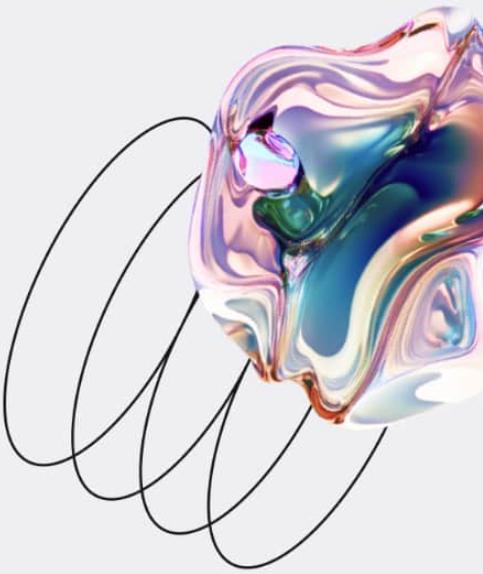
GeekBrains



Spring Testing Junit и Mockito для написания тестов

Фреймворк Spring

Оглавление

Введение

Привет, дорогие будущие мастера Spring!

Помните, как мы углублялись в мир Spring MVC, изучая, как создавать веб-приложения? Или когда мы работали с базами данных, используя Spring Data? А что насчет того сложного, но интересного момента, когда мы познакомились с Spring Security и узнали, как обеспечивать безопасность наших приложений? И, конечно же, нельзя забыть наш погружение в Spring AOP, когда мы учились перехватывать и модифицировать поведение наших приложений.

И вот, со всем этим багажом знаний, вы, возможно, задаетесь вопросом: “Как же мне убедиться, что все эти крутые вещи, которые я написал, работают правильно? Как мне спать спокойно, зная, что мой код безопасен и эффективен?” Ответ прост: тестирование!

Сегодняшний урок посвящен Spring Testing - магическому инструменту, который позволяет нам протестировать каждую часть нашего приложения, будь то контроллеры, службы или даже аспекты безопасности. Мы узнаем, как с помощью JUnit и Mockito создавать тесты, которые помогут нам быть уверенными в нашем коде.

Так что если вы когда-нибудь мечтали стать детективом, сегодня ваш шанс! Мы будем искать баги, находить их и устранять, делая наше приложение еще лучше!

Термины, используемые в лекции

Spring Test Context Framework — инфраструктура Spring для тестирования Spring-компонентов с JUnit или TestNG.

@SpringBootTest — аннотация, используемая для указания, что тестовый класс является тестом Spring Boot. Это помогает в автоматической настройке Spring Application Context.

JUnit — это библиотека Java для юнит-тестирования.

@Test — аннотация JUnit, указывающая, что метод является тестовым методом.

Mockito — популярная библиотека для создания mock-объектов в Java.

Mock — созданный объект, который имитирует поведение реального объекта. Используется в юнит-тестировании для изоляции тестируемого компонента.

@Mock и @MockBean — аннотации Mockito, используемые для создания и внедрения mock-объектов.

@InjectMocks — аннотация Mockito для создания экземпляра класса и внедрения в него mock-объектов.

@BeforeEach и @AfterEach — аннотации JUnit для методов, которые должны выполняться перед и после каждого тестового метода соответственно.

@BeforeAll и @AfterAll — аннотации JUnit для методов, которые должны выполняться один раз перед первым тестовым методом и после последнего тестового метода класса соответственно.

assertThat() — метод, используемый для утверждений в тестах.

Matchers — классы или методы, предоставляемые библиотеками тестирования (например, Mockito или Hamcrest), чтобы помочь в формировании утверждений.

Test Runner — класс или инструмент, который управляет запуском тестов и обеспечивает обратную связь о результатах.

ApplicationContext — центральный интерфейс для доступа к конфигурации приложения в Spring.

@ContextConfiguration м аннотация, используемая для указания конфигурации ApplicationContext для тестов.

@ActiveProfiles — аннотация для установки активных профилей Spring в тестах

История тестирования программного обеспечения

Друзья, вы когда-нибудь задумывались, почему мы так усердно тестируем свой код? Давайте вернемся немного назад во времени и узнаем, как всё началось.

В начале компьютерной эры, когда компьютеры были размером с комнату, и их задачей было выполнить несколько математических операций, тестирование не было таким актуальным. Программы были относительно простыми, и их легко было проверить вручную. Но по мере развития технологий, программы становились все сложнее и многозадачнее, и “надеяться, что всё будет ок” уже не катило.

К середине 20-го века, с ростом сложности ПО и появлением первых операционных систем, стало понятно, что нужен какой-то способ проверки качества программ. Ошибки в коде могли привести к дорогостоящим последствиям, будь то финансовые потери или даже угроза жизни, в случае промышленного или медицинского оборудования.

Так, в 1960-х годах, с появлением методологий разработки ПО, таких как водопадная модель, тестирование стало важным этапом в процессе разработки. Оно обеспечивало подтверждение того, что ПО соответствует требованиям и работает корректно.

А затем пришли методологии Agile и DevOps, которые сделали тестирование не просто этапом, но и неотъемлемой частью процесса разработки. Теперь тестирование стало регулярным, автоматизированным и постоянным. Современные инструменты, такие как JUnit и Mockito, которыми мы сегодня будем заниматься, делают тестирование быстрым, эффективным и надежным.

Баги — это не просто недоразумения, которые можно легко исправить. Иногда они могут стоить компаниям миллионов долларов, портить репутацию или даже вызывать катастрофы. Давайте посмотрим на несколько известных случаев, чтобы понять, насколько важно качественное тестирование.

1. **Шаттл “Маринер-1”**: В 1962 году НАСА потеряло космический корабль стоимостью 80 миллионов долларов из-за одной неверной точки в коде. Корабль ушел с курса и был уничтожен спустя 293 секунды после запуска. Кто бы мог подумать, что одна маленькая ошибка в коде приведет к таким последствиям?

2. **Amazon Web Services, 2017**: В один не такой уж прекрасный день, часть интернета просто “упала”. Множество сайтов и сервисов, таких как Slack, Trello и даже часть Apple services, столкнулись с проблемами. Причина? Один из инженеров AWS ввел неверную команду, пытаясь отладить систему.

3. **Knight Capital Group**: Эта торговая компания потеряла 440 миллионов долларов за 45 минут из-за ошибки в своей торговой системе. Эта ошибка стала причиной крупных потерь и почти разрушила компанию.

4. **Ошибки в играх**: Игры также не застрахованы от багов. Например, в первой версии игры “Assassin’s Creed” персонажи часто “падали” сквозь мир или

сталкивались с другими странными проблемами. Все это привело к снижению рейтинга игры и разочарованию игроков.

Подводя итог, каждая из этих ошибок могла бы быть предотвращена с помощью качественного тестирования. Баги - это не просто ошибки в коде. Это потенциальные угрозы вашему бизнесу, репутации и доверию пользователей. Так что давайте не пренебрегать тестами, чтобы избежать таких катастроф в будущем!

Виды тестов

Друзья, вы знали, что в мире разработки ПО существует множество разных видов тестов? Да, это не просто “тестирование”. Это целый арсенал инструментов, которые работают вместе, чтобы убедиться, что наше приложение работает, как ожидается. Давайте кратко рассмотрим основные из них и поймем, как они дополняют друг друга.

1. **Модульные тесты (Unit Tests)**: Эти тесты фокусируются на самых маленьких частях нашего кода, таких как функции или методы. Они убеждаются, что каждый “кирпичик” нашего ПО работает правильно в изоляции.

2. **Интеграционные тесты**: Здесь мы проверяем, как эти “кирпичики” работают вместе. Например, правильно ли наше приложение взаимодействует с базой данных или другими службами.

3. **Системные тесты**: Эти тесты проверяют работу всего приложения в целом. Они удостоверяются, что все компоненты совместно работают корректно и обеспечивают нужный результат для конечного пользователя.

4. **Тесты приемки**: Тут мы проверяем, соответствует ли наше ПО бизнес-требованиям и ожиданиям пользователей.

5. **Нагрузочные тесты**: Эти тесты помогают нам понять, как наше приложение справляется с большим количеством запросов или большим объемом данных.

Видите, каждый вид тестирования дополняет друг друга. Пока модульные тесты убеждаются, что каждый компонент работает правильно, интеграционные тесты проверяют, что эти компоненты хорошо взаимодействуют между собой. Системные тесты обеспечивают общую рабочую среду, а нагрузочные тесты гарантируют, что наше приложение готово к реальной жизни.

По сути, это как оркестр. Каждый музыкант может играть свой инструмент отдельно, но только вместе они создают гармоничную мелодию. И так же, как в оркестре, нам нужны все эти тесты, чтобы убедиться, что наше ПО играет без “ложных нот”.

Unit-тесты

Прежде чем идти дальше, давайте окунемся в удивительный мир модульного тестирования или, как их еще называют, unit-тестов. Почему они так важны? Почему они называются “модульными”? И как же их писать? Все это сейчас и узнаем.

Что такое unit-тесты?

Unit-тесты — это тесты, написанные для проверки функциональности отдельных частей программы, таких как функции, методы или классы. В идеале, каждый unit-тест должен быть изолированным и проверять только одну вещь.

Почему они так важны?

1. **Быстрая обратная связь**: Поскольку unit-тесты проверяют маленькие части кода, они выполняются быстро, что позволяет разработчикам мгновенно узнать о проблемах.
2. **Повышение уверенности**: Если ваши unit-тесты зеленые, это означает, что мелкие кирпичики вашего ПО работают, как ожидается.
3. **Упрощение изменений**: Быстро узнавая об ошибках, можно без страха вносить изменения в код.

Как писать unit-тесты?

1. **Изолированность**: Каждый тест должен быть независимым от других тестов и окружения. Это означает, что тест не должен полагаться на внешние сервисы, базы данных и т.д.
2. **Одна проверка на тест**: Хотя может быть искушение проверить несколько вещей в одном тесте, лучше сосредоточиться на одной конкретной проверке, чтобы было понятно, что именно пошло не так, если тест не прошел.

3. **Читаемость**: Тесты часто служат документацией к коду, поэтому их должно быть легко читать и понимать.
4. **ARR (Arrange, Act, Assert)**: Это общепринятый подход к структуре теста. Сначала устанавливаются начальные условия (Arrange), затем выполняется действие, которое вы хотите проверить (Act), и в конце производится проверка ожидаемого результата (Assert).

Пример (без привязки к конкретному языку):

Представьте функцию, которая складывает два числа:

```
function add(a, b):
  return a + b
```

Unit-тест для этой функции может выглядеть так:

```
test function test_addition():
  // Arrange
  input1 = 2
  input2 = 3
  expected_result = 5

  // Act
  result = add(input1, input2)

  // Assert
  assert result == expected_result
```

Unit-тесты - это ваш первый защитный барьер против багов, инструмент для поддержания качества кода и незаменимый помощник при рефакторинге.

Unit-тесты в Java

Первые шаги: JUnit

Когда говорят о unit-тестировании в Java, первое, что приходит в голову, — это JUnit. Этот фреймворк, созданный Кентом Беком и Эрихом Гамма в конце 90-х, стал де-факто стандартом для написания unit-тестов на Java.

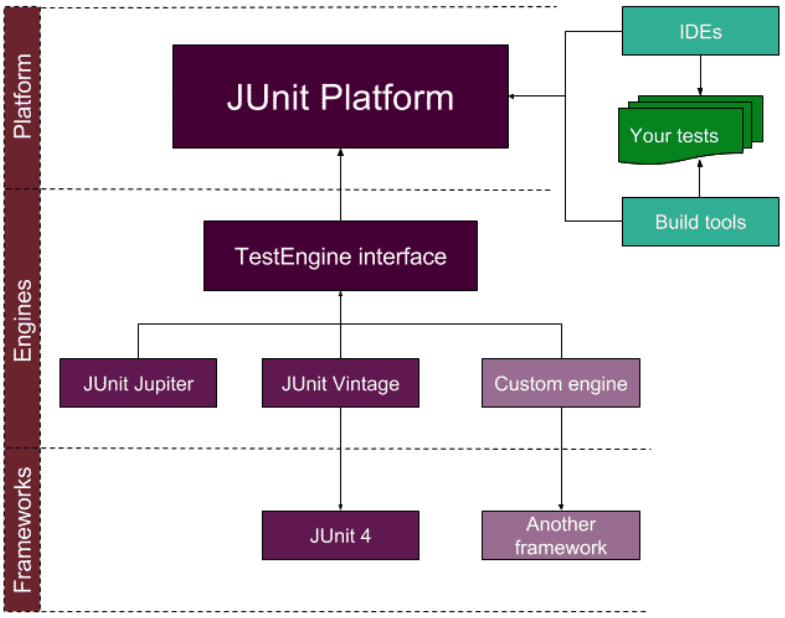
Изначально JUnit был разработан для Java и стал частью большого семейства фреймворков xUnit, каждый из которых предназначен для своего языка программирования. Благодаря простому и интуитивно понятному API, JUnit быстро стал популярным среди разработчиков.

Эволюция JUnit: От JUnit 3 до JUnit 5

1. **JUnit 3**: Первые версии JUnit основывались на именовании методов (все тестовые методы начинались с test). Хотя это было просто, структура тестов могла стать громоздкой.

2. **JUnit 4**: Введение аннотаций! Это принципиально изменило способ написания тестов. Теперь, чтобы указать тестовый метод, достаточно было добавить @Test перед ним. Это улучшило читаемость кода и добавило гибкости, позволив вводить такие возможности как параметризованные тесты, правила и многое другое.

3. **JUnit 5**: Современная итерация JUnit, представленная в 2017 году, принесла множество улучшений, в том числе модульную структуру. Это позволило разделить JUnit на три подпроекта: JUnit Platform, JUnit Jupiter и JUnit Vintage. Эта версия добавила много новых функций, таких как динамические тесты, расширенные возможности параметризации и более продвинутые аннотации.



Также рядом: TestNG, Spock и другие

Хотя JUnit является самым известным фреймворком для unit-тестирования в Java, существуют и другие инструменты, такие как TestNG и Spock. Эти инструменты предоставляют свои уникальные особенности и подходы к тестированию, предоставляя разработчикам больше возможностей.

В заключении, Java всегда была в авангарде разработки ПО, и в контексте модульного тестирования она дала миру множество инструментов и лучших практик. Будьте уверены, что в ваших руках мощные инструменты для обеспечения качества вашего кода!

Spring и Unit-тесты

Если вы когда-либо пытались написать unit-тесты для приложения на Spring, то знаете, что иногда это может быть довольно сложно из-за всей магии, которую делает этот фреймворк. Но у меня для вас хорошие новости: Spring обладает замечательными инструментами для упрощения процесса тестирования!

1. Spring TestContext Framework

Одной из ключевых фич Spring является его TestContext Framework. Этот фреймворк предоставляет аннотации и утилиты для загрузки Spring ApplicationContext в ваших тестах, что позволяет вам тестировать Spring-компоненты в изолированной среде.

2. @SpringBootTest

С помощью аннотации @SpringBootTest вы можете легко загрузить весь контекст вашего приложения или его части. Она автоматически настраивает ваше приложение и предоставляет все необходимое для тестирования.

3. Mocking с Mockito и @MockBean

Spring и Mockito идут рука об руку, когда дело доходит до написания unit-тестов. Spring предоставляет аннотацию @MockBean, которая позволяет вам создать mock-версию вашего бина и внедрить ее в контекст Spring. Это особенно полезно,

когда вы хотите изолировать конкретный компонент от внешних зависимостей, таких как репозитории или сервисы.

Пример:

```
@SpringBootTest
public class MyServiceTest {

    @Autowired
    private MyService myService;

    @MockBean
    private MyRepository myRepository;

    @Test
    public void testMyService() {
        when(myRepository.findSomething()).thenReturn(someData);

        // Некоторые проверки для myService
    }
}
```

4. @WebMvcTest и @DataJpaTest

В Spring Boot есть специальные аннотации для тестирования конкретных частей вашего приложения. Например, с @WebMvcTest вы можете тестировать только веб-слои вашего приложения, игнорируя остальное. А с @DataJpaTest можно сфокусироваться исключительно на тестировании JPA репозиториев.

5. @TestConfiguration

Определяет дополнительные конфигурационные классы или бины для теста. Эта аннотация помогает определить локальные компоненты, которые будут использоваться только во время тестирования.

Пример:

```
@TestConfiguration
public class MyTestConfiguration {
    @Bean
    public MyService myService() {
        return new MyMockedService();
    }
}
```

6. @BeforeAll и @AfterAll

Использование: Эти аннотации из JUnit 5 используются для определения методов, которые выполняются перед началом всех тестов и после их завершения.

Пример:

```
@SpringBootTest
public class MyTests {
    @BeforeAll
    public static void init() {
        // Некоторая инициализация
    }

    @AfterAll
    public static void cleanup() {
        // Очистка после тестов
    }
}
```

Spring действительно делает процесс написания unit-тестов проще и более интуитивным. Благодаря встроенным инструментам и интеграции с популярными инструментами, такими как Mockito, вы можете удостовериться, что ваше приложение на Spring надежно и готово к любым вызовам!

Интеграционные тесты

Давайте поговорим об интеграционных тестах. Что это за зверь такой и почему он так важен?

1. Что такое интеграционные тесты?

Интеграционные тесты — это тесты, которые проверяют взаимодействие между различными частями вашего приложения. В отличие от unit-тестов, которые фокусируются на тестировании отдельных модулей или функций, интеграционные тесты обеспечивают проверку того, как эти модули работают вместе.

2. Зачем они нужны?

Представьте себе огромную мозаику из тысячи кусочков. Если каждый кусочек идеально выполнен (это как хорошо написанный модуль или функция), но они плохо сочетаются друг с другом, то итоговая картина будет неудачной. Интеграционные тесты позволяют убедиться, что все кусочки гармонично соединены.

3. Какие аспекты они проверяют?

- **Взаимодействие с базами данных**: Как ваше приложение обращается к базе? Правильно ли данные записываются и извлекаются?
- **Общение с внешними сервисами**: Если ваше приложение подключается к сторонним API или другим службам, интеграционные тесты могут проверить, правильно ли это происходит.
- **Взаимодействие между модулями**: Как одна часть вашего приложения взаимодействует с другой?

4. Какие инструменты используются?

Для интеграционных тестов часто используются специализированные инструменты, которые могут имитировать внешние службы, базы данных или другие части системы. Эти инструменты могут быть частью тестового фреймворка или предоставляться отдельно.

5. В чем сложности?

- **Настройка и поддержка**: Так как интеграционные тесты взаимодействуют с различными частями системы, они могут быть сложнее в настройке.
- **Скорость выполнения**: Они обычно выполняются дольше, чем unit-тесты, так как тестируют более сложные взаимодействия.
- **Неустойчивость**: Из-за зависимости от множества компонентов интеграционные тесты могут быть менее стабильными.

Интеграционные тесты — ключевая часть стратегии тестирования. Они позволяют убедиться, что разные части вашего приложения работают синхронно и гармонично. Хотя они могут требовать дополнительных усилий по сравнению с unit-тестами, они являются неотъемлемой частью создания надежного программного обеспечения.

История интеграционных тестов

1. Ранние годы Java

Когда Java была представлена в 1995 году, основной акцент был сделан на простоту и переносимость кода. Но как и многие новые технологии того времени, она не имела широкого арсенала инструментов для тестирования. Разработчики опирались на ручное тестирование и простые скрипты для проверки своего кода.

2. Появление JUnit

В конце 1990-х годов JUnit появился как революционный инструмент для unit-тестирования в Java. Хотя первоначально он был ориентирован на тестирование отдельных модулей, с течением времени он стал основой для разработки интеграционных тестов, благодаря своей гибкости и расширяемости.

3. Рост популярности Maven и Ant

В начале 2000-х годов инструменты сборки, такие как Maven и Ant, стали популярными в Java-сообществе. Они предоставили структурированный способ

управления жизненным циклом проекта, включая фазы тестирования. С их помощью интеграционные тесты стали более автоматизированными и частью процесса сборки.

4. Появление инструментов для интеграционного тестирования

После того как основа была заложена с JUnit, Maven и Ant, стали появляться специализированные инструменты для интеграционного тестирования в Java. Инструменты, такие как Arquillian, позволяли разработчикам тестировать взаимодействие кода Java с различными серверами приложений, базами данных и другими внешними компонентами.

5. Современные тенденции

На протяжении последних лет Java продолжает быть в авангарде интеграционного тестирования. Широкий спектр инструментов, фреймворков и практик доступен для разработчиков, чтобы обеспечить качество и надежность своих Java-приложений.

Интеграционное тестирование в Java прошло долгий путь с момента своего зарождения. С развитием языка и технологий разработчики получили множество мощных инструментов и практик для обеспечения качества своих приложений. Вероятно, развитие в этой области продолжится, и будущее обещает быть еще более интересным!

Интеграционные тесты на Spring

Spring прекрасно работает с JUnit, что делает написание и запуск интеграционных тестов проще. Вам не придется изобретать колесо, всё уже здесь, готово к использованию! Для этого у него есть следующие компоненты:

1. Spring TestContext Framework

Spring предоставляет TestContext Framework, который обеспечивает функциональности для загрузки контекста Spring и кэширования его между тестовыми выполнениями для повышения производительности. Это очень удобно, когда у вас есть тяжелые интеграционные тесты, работающие с реальной БД или

другими внешними ресурсами.

2. @SpringBootTest

Аннотация @SpringBootTest автоматически конфигурирует ваше приложение, поднимает встроенный сервер и готовит всё для интеграционного тестирования. Она создает реальный контекст вашего приложения, что позволяет тестировать его так, как если бы оно работало в продакшене.

3. Встроенная поддержка баз данных

Spring позволяет легко настраивать встраиваемые базы данных, такие как H2, HSQL или Derby, для тестирования. Это удобно, потому что не требует отдельной конфигурации или настройки реальной базы данных.

4. MockMvc для веб-тестирования

Если вы разрабатываете веб-приложение, MockMvc станет вашим лучшим другом. Он позволяет быстро и легко создавать запросы к вашему приложению и проверять ответы, не запуская реальный сервер.

5. Spring Boot Test Slices

Spring Boot предлагает возможность “срезов тестов” (Test Slices), которые позволяют тестировать только определенные части вашего приложения, автоматически настраивая нужные компоненты. Например, @DataJpaTest настроит только слой JPA вашего приложения для тестирования.

Интеграционное тестирование в Spring — это настоящее удовольствие благодаря огромному арсеналу инструментов и возможностей, которые он предоставляет. И лучшая часть? Все эти инструменты разработаны так, чтобы упростить вашу жизнь и сделать процесс тестирования как можно более эффективным.

Нагрузочное тестирование

Нагрузочное тестирование (НТ) — это процесс проверки системы или компонента на его способность управлять заданной нагрузкой, обычно с учетом ожидаемого числа пользователей и операций в определенное время.

Зачем нам это?

- **Производительность**: НТ помогает удостовериться, что ваше приложение будет работать быстро, даже при большой нагрузке.
- **Стабильность**: Мы хотим узнать, не упадет ли наше приложение или не будет ли “тормозить”, когда много пользователей пытаются им пользоваться одновременно.
- **Масштабируемость**: Узнайте, насколько легко добавлять ресурсы (например, память, процессоры или серверы) для обработки дополнительной нагрузки.

Как провести нагрузочное тестирование?

1. **Определите ожидаемую нагрузку**: Начните с определения, сколько пользователей вы ожидаете на своем сайте или приложении в пиковые часы.
2. **Создайте тестовые сценарии**: Определите, какие операции пользователи будут чаще всего выполнять. Это может быть, например, поиск, добавление товара в корзину или просмотр видео.
3. **Подготовьте тестовое окружение**: Убедитесь, что ваша тестовая среда максимально приближена к реальной. Это может включать в себя настройку серверов, баз данных и сети.
4. **Выполните тесты**: Используйте специализированные инструменты для нагрузочного тестирования, чтобы имитировать действия многих пользователей одновременно.
5. **Анализируйте результаты**: После тестирования изучите данные, чтобы выявить узкие места или проблемы производительности, и определить, соответствует ли система вашим требованиям.

Отличия от других видов тестирования

НТ отличается от других типов тестирования, таких как функциональное или интеграционное, тем, что его основной целью является определение производительности, а не нахождение ошибок или багов.

Нагрузочное тестирование — неотъемлемая часть процесса разработки, которая помогает гарантировать, что ваше приложение будет работать гладко и без проблем, даже при большом количестве пользователей. Это вложение времени и ресурсов, которое, безусловно, окупится в будущем, когда ваш продукт стоит перед реальной аудиторией!

Нагрузочное тестирование в контексте Java

Раз уж мы уже поговорили о нагрузочном тестировании, давайте рассмотрим его в контексте нашего любимого языка — Java.

Когда Sun Microsystems выпустила Java в середине 90-х, акцент делался на переносимости и безопасности. Однако с ростом популярности языка, особенно в сфере корпоративных приложений и веб-сервисов, стало очевидно, что производительность и масштабируемость также играют ключевую роль.

С ростом интереса к производительности в Java-сообществе начали появляться первые инструменты для нагрузочного тестирования. Одним из первых и наиболее популярных инструментов стал **JMeter** от Apache. Первоначально он был разработан для тестирования веб-приложений, но со временем стал универсальным инструментом для нагрузочного тестирования.

Со временем, кроме JMeter, было разработано множество других инструментов, таких как **Grinder**, **Gatling** и многие другие. Эти инструменты позволяют разработчикам имитировать большое количество виртуальных пользователей, генерирующих нагрузку на приложения, написанные на Java.

Помимо них, Java предоставляет инструменты такие как **JVisualVM**, **Java Mission Control** и **Java Flight Recorder**. Это позволяет разработчикам не только генерировать нагрузку, но и детально анализировать, как их приложение реагирует на эту нагрузку.

С ростом микросервисных архитектур и облачных технологий важность нагрузочного тестирования в Java продолжает расти. Современные инструменты и платформы, такие как **Kubernetes** и **Istio**, предоставляют еще больше возможностей для тестирования и мониторинга приложений на Java в реальных условиях.

Java, начав как простой язык программирования, развился в мощную платформу, предлагающую разработчикам все необходимые инструменты для тестирования и оптимизации их приложений. И, хотя инструменты и подходы могут меняться, цель остается неизменной: обеспечить пользователям высокопроизводительные и надежные приложения.

Нагрузочное тестирование с помощью JMeter

Apache JMeter — это open-source инструмент, разработанный для тестирования производительности и функционального тестирования. Хотя он изначально разрабатывался для тестирования веб-приложений, сейчас JMeter может быть использован для различных сценариев тестирования.

Установка JMeter

1. Скачайте последний релиз JMeter с официального сайта.
2. Распакуйте архив в удобное место.
3. Запустите JMeter с помощью файла jmeter.bat (для Windows) или jmeter.sh (для Linux/macOS).

Создание тестового плана

1. Откройте JMeter.
2. Создайте новый тестовый план: Файл -> Новый.
3. Добавьте поток пользователей (Thread Group): ПКМ на вашем тестовом плане -> Добавить -> Потоки (Users) -> Thread Group.

Настройка запросов

1. В потоке пользователей добавьте HTTP-запрос: ПКМ на Thread Group -> Добавить -> Запросы (Requests) -> HTTP Request.
2. Введите детали вашего запроса (например, URL вашего сайта).

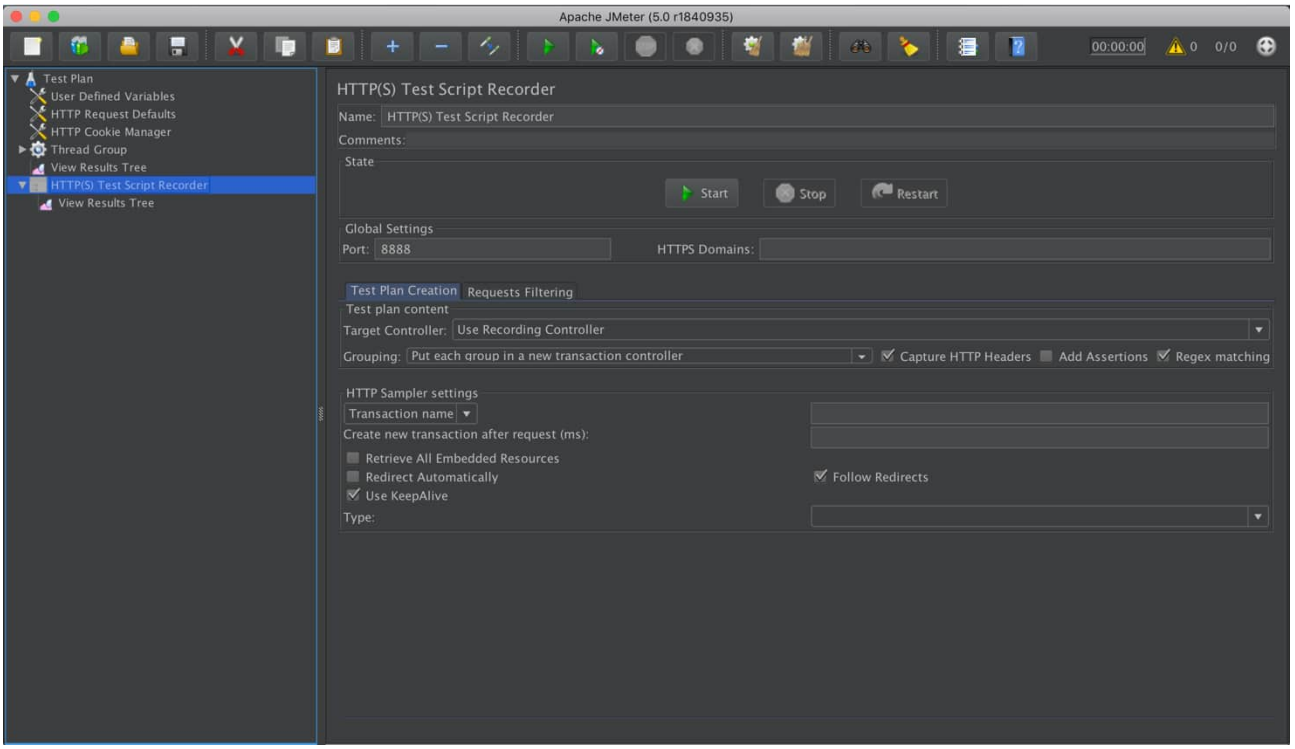
Добавление слушателей

Слушатели используются для просмотра результатов теста.

1. Добавьте слушателя: ПКМ на Thread Group -> Добавить -> Слушатели (Listeners) -> График результатов.

Запуск теста

Нажмите “▶️” (зеленую кнопку запуска) на панели инструментов JMeter. После завершения теста, вы можете просмотреть результаты в вашем слушателе.



Итак, мы только что создали базовый нагрузочный тест с помощью JMeter. Этот инструмент предлагает множество возможностей, включая возможность тестирования различных протоколов, подключение плагинов и интеграцию с CI/CD инструментами. Надеюсь, это было интересно и полезно!

Проект на Spring

Давайте создадим простое веб-приложение на Spring Boot. Это будет простой сервис “Заметки”, где пользователи могут добавлять, читать, редактировать и удалять свои заметки.

Начнём!

1. Создание проекта

Перейдите на Spring Initializr и выберите следующие настройки:

- Project: Maven
- Language: Java
- Packaging: Jar
- Java: 17

Dependencies: - Spring Web - Spring Data JPA - H2 Database

Затем скачайте проект и распакуйте его в удобное место.

2. Сущности

Создайте класс Note:

```
@Entity
public class Note {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @Lob
    private String content;

    // Геттеры, сеттеры и конструкторы
}
```

3. Репозиторий

Создайте интерфейс NoteRepository:

```
@Repository
public interface NoteRepository extends JpaRepository<Note, Long> {}
```

4. Сервис

Добавьте класс NoteService:

```
@Service
public class NoteService {

    @Autowired
    private NoteRepository noteRepository;

    public List<Note> getAllNotes() {
        return noteRepository.findAll();
    }

    public Note getNoteById(Long id) {
        return noteRepository.findById(id).orElse(null);
    }

    public Note saveOrUpdate(Note note) {
        return noteRepository.save(note);
    }

    public void deleteNote(Long id) {
        noteRepository.deleteById(id);
    }
}
```

5. Контроллер

Создайте NoteController:

```
@RestController
@RequestMapping("/api/notes")
public class NoteController {

    @Autowired
    private NoteService noteService;
```

```
    @GetMapping
    public List<Note> getAllNotes() {
        return noteService.getAllNotes();
    }

    @GetMapping("/{id}")
    public Note getNote(@PathVariable Long id) {
        return noteService.getNoteById(id);
    }

    @PostMapping
    public Note createNote(@RequestBody Note note) {
        return noteService.saveOrUpdate(note);
    }

    @PutMapping("/{id}")
    public Note updateNote(@PathVariable Long id, @RequestBody Note
updatedNote) {
        Note note = noteService.getNoteById(id);
        note.setTitle(updatedNote.getTitle());
        note.setContent(updatedNote.getContent());
        return noteService.saveOrUpdate(note);
    }

    @DeleteMapping("/{id}")
    public void deleteNote(@PathVariable Long id) {
        noteService.deleteNote(id);
    }
}
```

6. Настройка базы данных

Откройте application.properties и добавьте следующее:

```
spring.datasource.url=jdbc:h2:mem:notesdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
```

```
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true
```

Это позволит нам использовать встроенную базу данных H2.

7. Запуск

Теперь, когда у нас есть базовое приложение, вы можете запустить его, используя mvn spring-boot:run или из вашей IDE.

После запуска, ваш сервис будет доступен на http://localhost:8080/api/notes.

Таким образом, у нас есть простой сервис “Заметки” на Spring Boot. В следующих разделах мы рассмотрим, как написать различные виды тестов для этого сервиса.

Юнит-тестирование нашего проекта

Начнем с юнит-тестов. Эти тесты направлены на проверку отдельных, независимых компонентов вашего приложения (как правило, методов). В нашем случае мы начнем с тестирования службы NoteService.

Почему юнит-тесты здесь полезны?

Представьте юнит-тесты как мелкие охранники, которые стоят у каждого входа в ваш замок (приложение). Они обеспечивают базовую безопасность, удостоверяясь, что каждый отдельный метод функционирует правильно. Плюсы:

- **Быстрота**: Юнит-тесты запускаются быстро.
- **Изоляция**: Они проверяют функциональность в изоляции, исключая внешние зависимости.
- **Обнаружение ошибок**: При изменении кода юнит-тесты помогают быстро выявить проблемы.

Поехали!

1. Подготовка зависимостей

В pom.xml убедитесь, что у вас есть зависимости JUnit и Mockito:

```
<!-- JUnit -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<!-- Mockito -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>
```

2. Написание теста для NoteService

Создайте новый класс NoteServiceTest в папке src/test/java в том же пакете, что и NoteService.

```
@RunWith(SpringRunner.class)
public class NoteServiceTest {

    @InjectMocks
    private NoteService noteService;

    @Mock
    private NoteRepository noteRepository;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
```

```
    public void getAllNotesTest() {
        Note note = new Note();
        note.setTitle("Test Title");
        note.setContent("Test Content");

        List<Note> expectedNotes = Collections.singletonList(note);

        when(noteRepository.findAll()).thenReturn(expectedNotes);

        List<Note> actualNotes = noteService.getAllNotes();

        assertEquals(expectedNotes, actualNotes);
    }

    // ... Другие тесты для методов сервиса
}
```

Здесь мы делаем следующее:

- Инициализируем наш сервис с помощью @InjectMocks.
- Создаем “мок” для нашего репозитория с помощью @Mock.
- Используем when(...).thenReturn(...) чтобы настроить, что будет возвращать наш мок-репозиторий при вызове метода.
- Наконец, сравниваем ожидаемый результат и результат, который вернул наш сервис.

Юнит-тесты позволяют нам быть уверенными, что наши отдельные методы или компоненты работают правильно. В контексте нашего приложения это может означать, что мы знаем, что наш сервис правильно взаимодействует с репозиторием, даже если мы подменяем реальный репозиторий на мок.

Интеграционное тестирование нашего проекта

Друзья, переходим к интеграционным тестам! Если юнит-тесты — это мелкие охранники у каждого входа в наш замок, то интеграционные тесты проверяют, как эти входы работают вместе, когда появляются посетители.

Почему интеграционные тесты полезны для нас?

- **Проверка взаимодействия**: Они убедятся, что разные части вашего приложения работают вместе без ошибок.
- **Контекст приложения**: В отличие от юнит-тестов, интеграционные тесты запускаются в реальном контексте приложения.
- **База данных**: Мы можем использовать встроенные базы данных, такие как H2, чтобы эмулировать реальную работу с данными.

Погружаемся!

1. Настройка тестового окружения

Добавьте зависимость H2 Database в pom.xml:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

2. Написание теста для NoteService

В папке src/test/java создайте новый класс NoteServiceIntegrationTest.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class NoteServiceIntegrationTest {

    @Autowired
```

```
    private NoteService noteService;

    @Autowired
    private NoteRepository noteRepository;

    @Before
    public void setUp() {
        // Очищаем базу данных перед каждым тестом
        noteRepository.deleteAll();
    }

    @Test
    public void getAllNotesIntegrationTest() {
        Note note = new Note();
        note.setTitle("Integration Test Title");
        note.setContent("Integration Test Content");

        noteRepository.save(note);

        List<Note> notes = noteService.getAllNotes();
        assertTrue(notes.size() > 0);
        assertEquals(note.getTitle(), notes.get(0).getTitle());
    }

    // ... Другие интеграционные тесты
}
```

Здесь:

- @SpringBootTest говорит Spring загрузить полный контекст приложения.
- Мы используем реальные NoteService и NoteRepository.
- В setUp мы очищаем базу данных перед каждым тестом.

Интеграционные тесты дадут нам уверенность в том, что разные части нашего приложения (как сервис и репозиторий) работают вместе, как ожидалось. Для нашего проекта это означает, что когда мы сохраняем или извлекаем заметки, все работает гладко, без каких-либо препятствий или ошибок.

Нагрузочное тестирование нашего Spring-проекта

Перед тем как отпустить наш сервис в “дикий мир”, давайте убедимся, что он не упадёт от первого же порыва ветра (читай: внезапного потока пользователей). Нагрузочное тестирование призвано проверить, как наш сервис поведёт себя при различных объемах трафика.

Для нагрузочного тестирования существует множество инструментов. Сегодня я предлагаю использовать один из самых популярных инструментов в мире Java - **Apache JMeter**.

Шаг 1. Установка JMeter

Скачайте и установите JMeter. После установки запустите JMeter.

Шаг 2. Создание тестового плана

1. Создайте новый **Test Plan**.
2. Добавьте **Thread Group**. Это ваша группа пользователей, которые будут осуществлять запросы.
3. Установите нужное количество потоков (например, 100 пользователей) и количество повторов (например, 10 раз).

Шаг 3. Добавление и конфигурация HTTP Request

1. Выберите ваш **Thread Group** и добавьте **Sampler -> HTTP Request**.
2. Настройте его:

– Server Name: localhost

– Port Number: 8080 (или тот порт, на котором у вас работает сервис)

– Path: /notes (если вы хотите получить все заметки)

– Method: GET

Шаг 4. Добавление Listener

Чтобы видеть результаты наших запросов, добавьте **Listener**. Самый простой и понятный — **View Results in Table**.

Шаг 5. Запуск теста

Запустите тест, кликнув на “Start” (зелёный треугольник). После завершения теста вы увидите результаты в вашем Listener’е.

Шаг 6. Анализ результатов

Смотрите на ответы. Особенно интересуют: — Сколько запросов завершились успешно (и были ли вообще ошибки)? — Сколько времени в среднем занимал каждый запрос?

JMeter — мощный инструмент, который может делать гораздо больше, чем просто отправлять GET-запросы. Но даже с этим базовым функционалом вы сможете убедиться в стабильности вашего сервиса. Если результаты вас не устраивают - это знак вернуться к коду и оптимизировать его!

Метрики нагрузочного тестирования

После того как мы провели нагрузочное тестирование нашего сервиса с помощью JMeter, у нас появилось куча данных. Но что делать с этой кучей цифр? Как понять, что всё в порядке или наоборот, пора звонить знакомому, который работает с Java-оптимизацией? Давайте разберёмся вместе!

1. Throughput (Пропускная способность)

Это количество запросов в секунду, которое ваш сервер может обработать.

Что это значит для нас?

- Если пропускная способность низкая, это может указывать на проблемы с производительностью.
- Если она резко падает при увеличении нагрузки, это точно тревожный знак.

2. Response Time (Время ответа)

Это время, которое требуется серверу, чтобы ответить на ваш запрос.

Что это значит для нас?

- Если время ответа высокое даже при небольшой нагрузке, то это может указывать на проблемы в определённых частях вашего приложения.
- Увеличение времени ответа при увеличении нагрузки - ещё один показатель проблем с производительностью.

3. Error Rate (Процент ошибок)

Процент запросов, которые вернули ошибку.

Что это значит для нас?

– Любое значение, отличное от 0%, уже плохо. Это может указывать на проблемы с кодом, базой данных или даже инфраструктурой.

– Если процент ошибок увеличивается при нагрузке, это явно не то, что мы хотели бы видеть.

4. Concurrent Users (Конкурентные пользователи)

Сколько пользователей одновременно взаимодействуют с вашим сервисом.

Что это значит для нас?

– Если приложение начинает “хромать” уже на небольшом числе пользователей, пора думать об оптимизации.

5. CPU/Memory Utilization (Использование ЦПУ/Памяти)

Какой процент ресурсов сервера используется при нагрузке.

Что это значит для нас?

– Если ваш сервер постоянно на пределе своих возможностей, даже если он справляется с текущей нагрузкой, при малейшем увеличении трафика он “упадет”.

Результаты нагрузочного тестирования — это не просто числа, это здоровье вашего приложения. Они могут показать вам слабые места, которые нужно усилить. Понимание этих метрик — ключ к успешной оптимизации и масштабированию вашего Spring-проекта.

Заключение

Мы пройденным материалом открыли для себя многие аспекты тестирования на платформе Spring. Вспомним, почему это так важно:

1. **Тестирование** — это больше, чем просто “убедиться, что всё работает”. Это залог уверенности в том, что при развитии и изменении вашего приложения оно будет стабильно и надёжно.

2. **Unit-тесты** дают нам уверенность в корректности отдельных модулей, а **интеграционные тесты** позволяют проверить взаимодействие между различными частями системы.
3. **Нагрузочное тестирование** — необходимый инструмент для оценки производительности вашего приложения и его готовности к масштабированию.

Итак, зная, как провести все эти виды тестов, вы обеспечиваете себя более качественным и стабильным кодом, что делает вас настоящим профессионалом в Java и Spring!

Теперь, когда вы знаете, как проверить работоспособность и производительность вашего приложения, пора узнать, как следить за его состоянием в реальном времени. В следующем уроке мы погрузимся в удивительный мир **Spring Actuator** и научимся настраивать мониторинг с использованием **Prometheus** и **Grafana**. Эти инструменты станут вашими глазами и ушами в живой экосистеме вашего приложения!

Спасибо за то, что были со мной на этом уроке. До встречи на следующем занятии, где мы сделаем еще один шаг к совершенству в мире Spring!

Что можно почитать еще?

1. Изучение Spring Boot 2.0. Грег Тернквист
2. Освоение Spring Boot 2.0. Динеш Раджпут

Используемая литература

1. Spring Boot в действии. Крейг Уоллс.
2. Spring Microservices в действии. Джон Карнелл
3. Cloud Native Java. Джош Лонг и Кенни Бастани