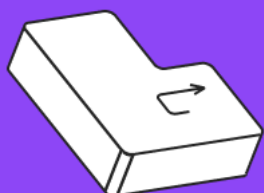




# Другие виды тестирования

Введение в юнит-тестирование



# Оглавление

Введение	3
Термины, используемые в лекции	3
Какое тестирование бывает	4
Юнит-тестирование	4
Минусы юнит-тестов	5
Интеграционное тестирование	6
Минусы интеграционных тестов	6
Плюсы интеграционных тестов	7
Отличия модульного и интеграционного тестов	7
Testcontainers	10
Интеграционное тестирование в Python	14
Сквозное тестирование	15
Selenium	17
Установка Selenium	19
Другие виды тестирования	20
Количество разных тестов в реальных проектах	21
Пирамида тестов	22
Автоматизация тестирования	23
Инструменты для автоматизации процессов CI	24
Подведем итоги	25
Домашнее задание	26
Контрольные вопросы	26
Полезные материалы	26

# Введение

Мы рассмотрели основные составляющие юнит-тестирования и для полноты картины нужно рассмотреть оставшиеся две составляющие тестирования программного обеспечения. На этом уроке особое внимание будет уделено интеграционному и сквозному тестированию. Мы рассмотрим, каким образом эти техники тестирования могут помочь вам улучшить качество разрабатываемого продукта и почему их выделяют в отдельные типы.

**На этой лекции вы узнаете про:**

- Интеграционное тестирование
- Сквозное тестирование
- Особенности и отличия
- Инструменты интеграционного тестирования
- Инструменты сквозного тестирования
- Автоматизацию тестирования

## Термины, используемые в лекции

**HTTP-протокол** — протокол прикладного уровня передачи данных, изначально - в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных

**Docker** — программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой *контейнеризации*, а также предоставляет набор команд для управления этими контейнерами

**Контейнеризация приложений** — это процесс «упаковки» приложения со всем его окружением и зависимостями в контейнер, который может быть развернут на любой Linux-системе.

# Какое тестирование бывает

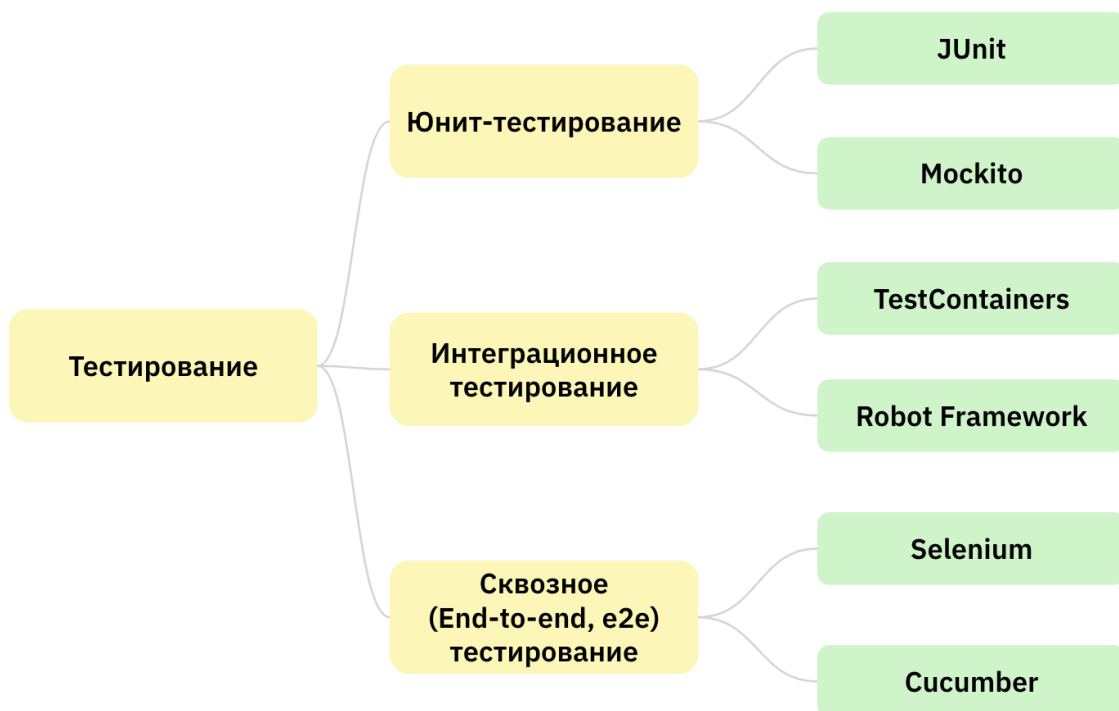
Юнит (модульное) и другие виды тестирования, которые мы рассмотрим являются составляющими функционального тестирования.

Функциональное тестирование (functional testing) рассматривает заранее указанное поведение и основывается на анализе спецификации компонента или системы в целом, т.е. проверяется корректность работы функциональности приложения.

**По степени изолированности тестов выделяют следующие виды тестов:**

- Юнит
- Интеграционное
- Сквозные

На схеме условно показано, что все эти тесты являются частью процесса тестирования. Там представлены одни из самых популярных инструментов для проведения разных типов тестов. С некоторыми мы уже знакомы, а другими познакомимся в этой лекции.



## Юнит-тестирование

В Java (и в остальных языках) есть несколько видов тестирования для разных целей. Мы уже изучили юнит-тестирование, оно служит для надежной и быстрой


проверки небольших модулей кода, и знаем про популярные инструменты тестирования Java:

- **JUnit** является одним из популярных фреймворков для тестирования модулей приложений на языке Java. Это помогает разработчикам проверить приложение на правильность работы, используя простые инструменты и инструкции.
- **Mockito** — это фреймворк поддержки тестовых двойников на языке Java. Он позволяет создавать не настоящие объекты, которые принимают и возвращают предусмотренные для них значения, а также проверяют, что методы объектов были вызваны с нужными параметрами.

## Минусы юнит-тестов

Все виды тестирования нужны, каждый решает конкретную задачу, которую не может решить другой вид тестирования.

1. **Они не проверяют взаимосвязь компонентов в рамках целой системы**, а лишь детально тестируют поведение каждого из модулей. Для полноценного тестирования приложения нужно объединить процессы интеграционного и модульного тестирования в единую сложную структуру. Кроме этого, тестирование модулей может быть очень продолжительным процессом, выделяющим значительное количество ресурсов.
2. Сложно тестировать **фронтенд**. Многие визуальные задачи, такие как тестирование графического интерфейса, почти невозможно выполнить с помощью инструментов юнит-тестирования.
3. **Много дополнительного кода** нужно постоянно поддерживать и редактировать код тестов, из-за большого объема таких тестов это может стать проблемой

 Можно сделать вывод, что юнит тестов недостаточно для полноценного тестирования приложения. Они являются полезным инструментом, но для обеспечения качества приложения необходимо выполнять и другие типы тестов.

# Интеграционное тестирование



Проверка компонентов в изоляции друг от друга важна, но не менее важно проверить, как эти компоненты работают **в интеграции друг с другом** и внешними системами.

Интеграционное тестирование помогает выявить проблемы, которые возникают при обмене данными различных частей приложения. **Оно является более сложным, чем unit-тестирование**, так как при его выполнении приходится частично воссоздавать реальную среду и поскольку для выполнения интеграционного тестирования необходимо предусмотреть работу всего приложения, данный процесс может требовать большего времени и ресурсов. Инструменты интеграционного тестирования в Java включают такие популярные фреймворки и библиотеки, как например:

- **TestContainers** — это платформа с открытым исходным кодом для предоставления одноразовых, облегченных экземпляров баз данных, брокеров сообщений, веб-браузеров или практически всего, что может запускаться в контейнере Docker.
- **Robot Framework** — это универсальная платформа автоматизации с открытым исходным кодом. Его можно использовать для автоматизации тестирования и автоматизации роботизированных процессов

## Минусы интеграционных тестов

1. Интеграционные тесты **длительно выполняются**. Так как окружение близко к реальному, то это дополнительные издержки по времени, например, развертывание баз данных занимает больше времени чем проведение многих юнит тестов
2. Второй недостаток — они **нестабильны** из-за окружения опять же, из-за сильных зависимостей извне. Допустим машине, на который вы проводите интеграционные тесты, может просто не хватить памяти
3. **Сложно локализовать ошибки**. Юнит-тесты можно настроить так, чтобы указывалась конкретная строка. А так как в интеграционном тестировании много компонентов, то сразу не понятно где ошибка. Также интеграционные тесты сложнее в написании.

🔥 Те же минусы относятся и к [e2e-тестам \(сквозным\)](#), они сильнее проявляются, что приводит к более высокой стоимости таких тестов. С этим и связано их небольшое количество относительно других тестов

## Плюсы интеграционных тестов

Интеграционное тестирование имеет преимущества:

- Первое — это тестирование, в реальном окружении (почти) вы сами контролируете что тестировать.
- Тестирование находит проблему до развертывания сборки на окружении.
- Так же тесты можно запустить в процессе разработки и автоматизировать их запуск

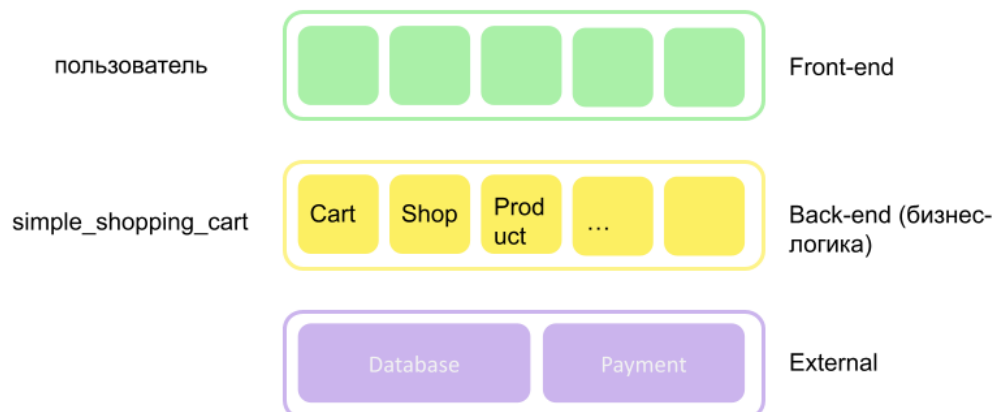
## Отличия модульного и интеграционного тестов

Рассмотрим написание тестов на примере абстрактного проекта:

Другие виды тестирования

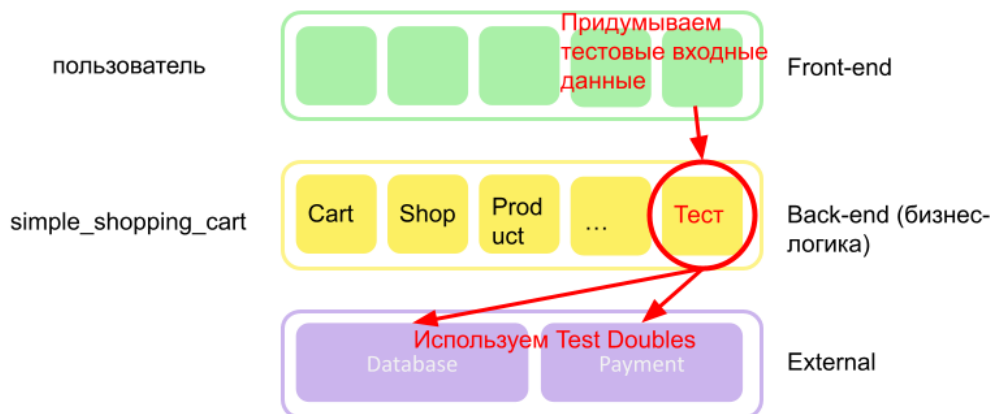


### Пример



На втором семинаре мы протестировали логику приложения магазина Simple shopping cart (по сути — бэкенда), которая обеспечивает работу приложения магазина. Для реального функционирования необходим также фронтенд — часть, видимая пользователю, а также подключаемые компоненты, такие как базы данных или платежные сервисы, и их взаимодействие с бэкендом тоже нужно проверять.

## Пример. Модульный тест



Представим взаимодействие бэкенда с платежной системы. В качестве входных параметров мы вручную подставляли придуманные значения руководствуясь некоторыми правилами, выбирали из граничных значений и старались предусмотреть все варианты развития. Для взаимодействия с базой данных мы могли бы использовать мок-объект, так мы полностью изолируем тест в бизнес-логике от остальных слоев.

Согласно определению **интеграционное тестирование** — это одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе.

Или более лаконичное определение — это тестирование, когда проверяются два или более компонентов системы. То есть это практически тот же модульный тест, но в почти реальном окружении, таком, в котором оказывается реальный пользователь.



В отличие от юнит-тестирования, нацеленного на (небольшие) модули, интеграционное тестирование является в некоем роде противоположностью — оно проверяет весь код и плюс его зависимости.

Юниты-модули могут быть сколь угодно идеальными, а приложение в целом будет бесполезным, если весь код «в целом» не работает как положено во внешнем окружении после деплоя. Другими словами, интеграционное тестирование тестирует приложение «во всей его красе».



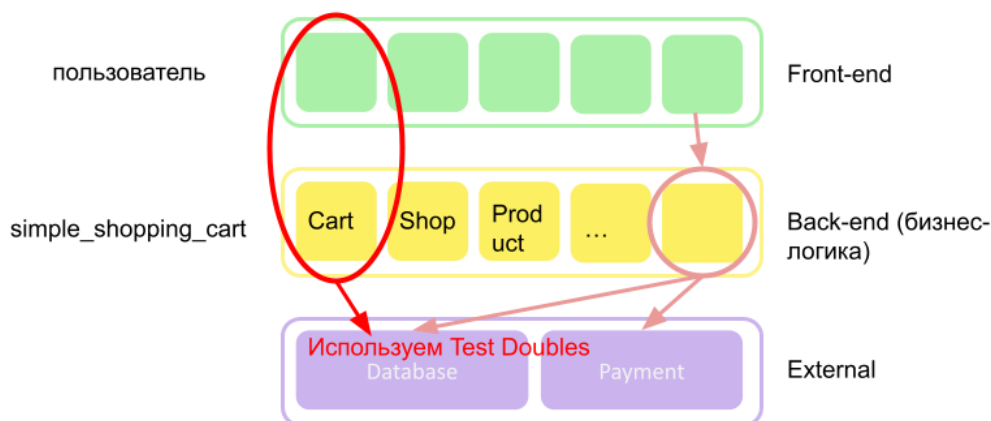
Интеграционные тесты выполняются QA-отделом (или QA-командой), который выполняет тест-кейсы, проверяя производительность и функциональность приложения.

🔥 Интеграционные тесты имеют ряд преимуществ по сравнению с модульными, включая обнаружение непредвиденных проблем между различными компонентами. Однако то, что делает их особенно полезными, также является их главным недостатком — они очень сложны в написании. В связи с этим, для экономии ресурсов, количество интеграционных тестов меньше, чем количество модульных

Другие виды тестирования



### Пример. Интеграционный тест



Представьте, что перед вами стоит задача протестировать другой тестовый сценарий: покупку товара в нашем магазине. Сценарий предполагает какое-то взаимодействие с пользователя с фронтендом, в результате которого фронтенд возвращает некоторые параметры бэкенду, например идентификаторы продуктов, которые выбрал пользователь, а бекенд взаимодействует с базой данных, отдаст ответ фронтенду о продуктах в корзине, о том, что можно их отображать пользователю.

Пример сценария для создания интеграционного теста:

1. Поднимается окружение, которое похоже (но не полностью!), на окружение реального пользователя
2. Создается тестовый двойник базы данных, чтобы не использовать реальное подключение. **Причем тест все еще будет считаться интеграционным, так как мы тестируем несколько компонентов!**

3. Далее уже знакомый нам процесс юнит-тестирования
4. При необходимости, можно воспользоваться инструментами отчета и другими для автоматизации процессов.

## Testcontainers

Рассмотрим популярный инструмент для интеграционного тестирования в java (и не только).

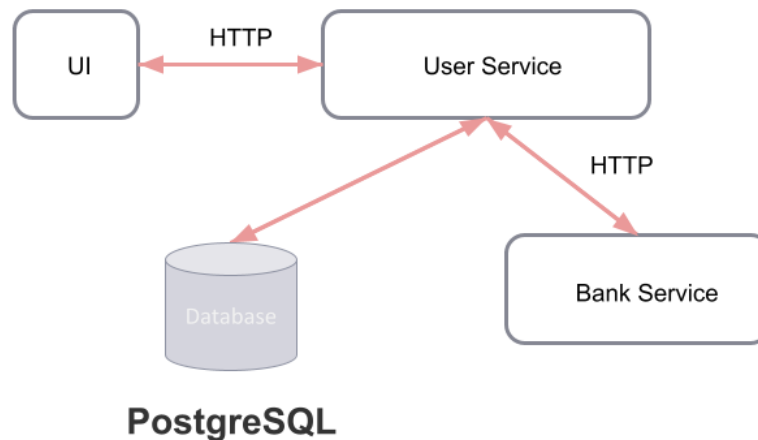
**TestContainers** — это библиотека которая поддерживает тесты JUnit и предоставляет легкие, временные экземпляры основных баз данных, веб-браузеров для Selenium или чего угодно еще, что можно запускать в Docker-контейнере, это по сути обертка вокруг java-библиотеки для docker.



Из интересных особенностей можно выделить:

- обнаружение окружений (win, mac, linux)
- чистка контейнера после завершения теста

## Testcontainers. Обзор проекта



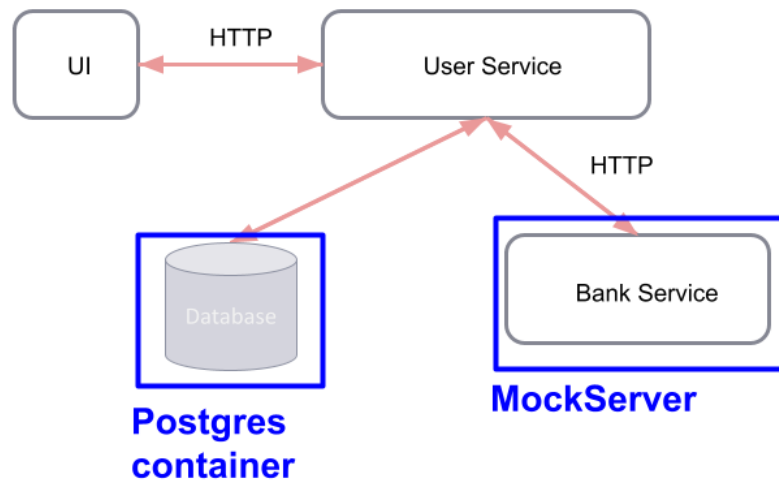
В качестве примера настройки и использования библиотеки рассмотрим небольшой финансовый проект, для которого мы хотим написать интеграционный тест с помощью библиотеки TestContainers.

Существует некий сервис который назовем UserService, который хранит информацию о пользователях банка. Используя этот сервис, пользователь может зайти и посмотреть информацию о себе.

У этого сервиса есть UI, таким образом сервис может общаться с пользователем через HTTP протокол. Получать какие-то данные и отображать их пользователю.

Сервис может обращаться к базе данных, где хранится информация о пользователе, это будет постгрес. Этот сервис также связан с другим сервисом, который назовем Bank Service, который хранит, информацию о кредитной истории пользователя, его тратах и так далее. Между собой сервисы взаимодействуют также по HTTP-протоколу.

## Testcontainers. Обзор проекта



Теперь определим, что будем делать с зависимостями в проекте.

- Постгрес мы можем заменить на **постгрес-контейнер**, для этого есть готовое решение в библиотеке.
- Для Bank Service мы можем использовать **МокСервер**, чтобы не включать реальный сервис в тестирование.

Мок Сервер это отдельная утилита, которая поднимает отдельный сервер, к которому вы можете обратиться по HTTP и можно определить правила по которым такой сервер будет возвращать ответ. В мок сервере запросы по HTTP реально отсылаются, то есть сервер максимально похож на реальный объект.

```
@Testcontainers
class MixedLifecycleTests {

    @Container
    private PostgreSQLContainer postgresqlContainer = new PostgreSQLContainer()
        .withDatabaseName("foo")
        .withUsername("foo")
        .withPassword("secret");

    @Test
    void test() {
        assertThat(postgresqlContainer.isRunning()).isTrue();
    }
}
```

Для настройки контейнеров с использованием JUnit 5 достаточно использовать аннотации.

→ Аннотация **@Testcontainers** используется над классом, и означает что вы будете использовать тест контейнеры.

→ Аннотация **@Container** показывает как создается сам контейнер.

В примере показано создание postgres-контейнера в виде базы данных с названием, и доступом по заданному логину и паролю ,ниже создается обычный тест, который мы уже создавали с Junit который проверяет, что контейнер запущен. Контейнер будет создаваться автоматически каждый раз при запуске тестового класса, каждый раз новый контейнер

Есть много поддерживаемых готовых модулей:

- JDBC support
- Cassandra Module
- Clickhouse Module
- MongoDB Module
- Postgres Module
- Docker Compose Module
- Elasticsearch container
- Kafka Containers
- Mockserver Module
- Nginx Module
- RabbitMQ Module
- и другие

```
@Container
    public static MockServerContainer mockServer = new
MockServerContainer(DockerImageName.parse("jamesdbloom/mockserver"));

    . . .
```

```

        public void initialize(ConfigurableApplicationContext
configurableApplicationContext) {
            TestPropertyValues.of(
                "service-bank.base-url=http://" +
mockServer.getContainerIpAddress() + ":",
mockServer.getFirstMappedPort()
            ).applyTo(configurableApplicationContext.getEnvironment());
        }
    }
}

```

Это часть настройки мок-сервера, где мы задаем адрес, по которому он будет доступен. (код представлен не полностью, так как для полноценного примера нужен спринг)

И сам тест, в котором сначала настраивается поведение мок-сервера так, чтобы он мог отвечать по запросу /users/1 и возвращал ответ в виде пустого массива [].

И проверяем с помощью restTemplate, что запрос дошел, и что вернулось в ответ, используя уже знакомый вам синтаксис утверждений:

```

@Test
void getCitizenStatistic_forHelenZheludOK() {
    new MockServerClient(mockServer.getHost(),
mockServer.getServerPort())
        .when(request()
            .withPath("/fines/1"))
        .respond(response()
            .withBody("[]").withHeader(CONTENT_TYPE,
APPLICATION_JSON_VALUE));

    ResponseEntity<CitizenStatisticDto> entity = restTemplate
        .getForEntity("http://localhost:" + port +
"/citizens/98y783y7438uyegbkyg37yrg", CitizenStatisticDto.class);

    assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(entity.getBody()).isNotNull();

    assertThat(entity.getBody().finesTotal).isEqualTo(BigDecimal.ZERO);
}

```

Testcontainers довольно популярная библиотека, которую используют многие крупные компании:

- JetBrains
- eBay Marketing
- JHipster
- Google
- Jenkins
- Elastic
- ...

## Интеграционное тестирование в Python

Интеграционное тестирование в Python можно осуществлять с помощью различных инструментов. Одним из самых популярных инструментов является **PyTest**.



Search  Go

About pytest

pytest is a mature full-featured Python testing tool that helps you write better programs.

Contents

- [Home](#)
- [Get started](#)
- [How-to guides](#)
- [Reference guides](#)

### pytest: helps you write better programs

The `pytest` framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries.

`pytest` requires: Python 3.7+ or PyPy3.

PyPI package name: [pytest](#)

### A quick example

```
# content of test_sample.py
def inc(x):
    return x + 1

def test_answer():
    assert inc(3) == 5
```

#### Next Open Trainings

- [Professional Testing with Python](#), via [Python Academy](#), March 7th to 9th 2023 (3 day in-depth training), Remote and Leipzig, Germany

Also see [previous talks and blogposts](#).

Источник: [Pytest](#)

PyTest — это фреймворк для автоматизации тестирования, который позволяет создавать, управлять и анализировать тест-кейсы. PyTest предоставляет удобный API, который позволяет упростить написание тест-кейсов.

Другим инструментом, который можно использовать для интеграционного тестирования, является **unittest**.

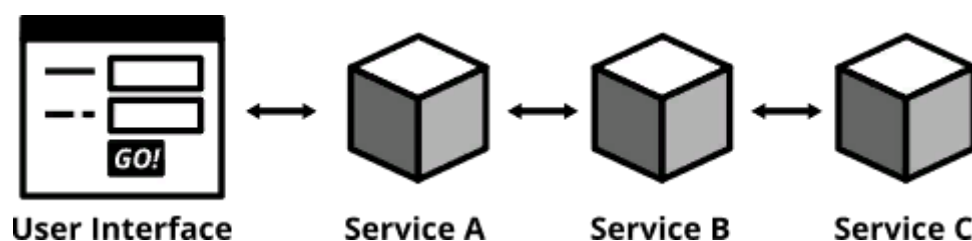
# Сквозное тестирование

**Сквозное тестирование (End-to-end, E2E, Chain testing)** — это вид тестирования, используемый для проверки программного обеспечения от начала до конца, отсюда и название «End-to-End», а также его интеграцию с внешними интерфейсами.

Цель сквозного тестирования состоит в проверке всего программного обеспечения на предмет зависимостей, целостности данных и связи с другими системами, интерфейсами и базами данных для проверки успешного выполнения полного производственного сценария.



По сравнению с интеграционными тестами такие тесты обычно включают пользовательский интерфейс, если он есть.



Источник: [Пирамида тестов на практике](#)

Сквозное тестирование обычно проводится после модульного и интеграционного тестирования. Для его проведения используются реальные данные и тестовая среда для имитации рабочего режима.

**Современные системы ПО сложны и взаимосвязаны с большим количеством подсистем, которые могут существенно отличаться от существующих систем. Вся система может разрушиться из-за отказа любой подсистемы, что представляет собой серьезный риск. Этого риска мы как раз и стремимся избежать с помощью сквозного тестирования.**

К инструментам сквозного тестирования можно отнести:

- **Selenium** — он позволяет автоматизировать тестирование веб-приложений. Это позволит тестировщикам проводить тестирование быстро, эффективно и надежно.

С использованием Selenium тестировщики могут эмулировать действия пользователей, такие как ввод данных, навигация по страницам и просмотр вывода.



Он также предоставляет удобный интерфейс для создания тестовых сценариев и инструмент, называемый WebDriver, для управления приложениями браузеров и запуска селениевских тестов.

Кроме того, Selenium очень популярный инструмент и включает в себя подключаемые модули, позволяющие тестировщикам работать с разными платформами и технологиями.

- **Cucumber** — это фреймворк автоматизации тестирования программного обеспечения с открытым исходным кодом для разработки, ориентированной на поведение.

Он использует понятный для бизнеса, специфичный для домена язык под названием Gherkin для определения поведения функций, которые становятся тестами.

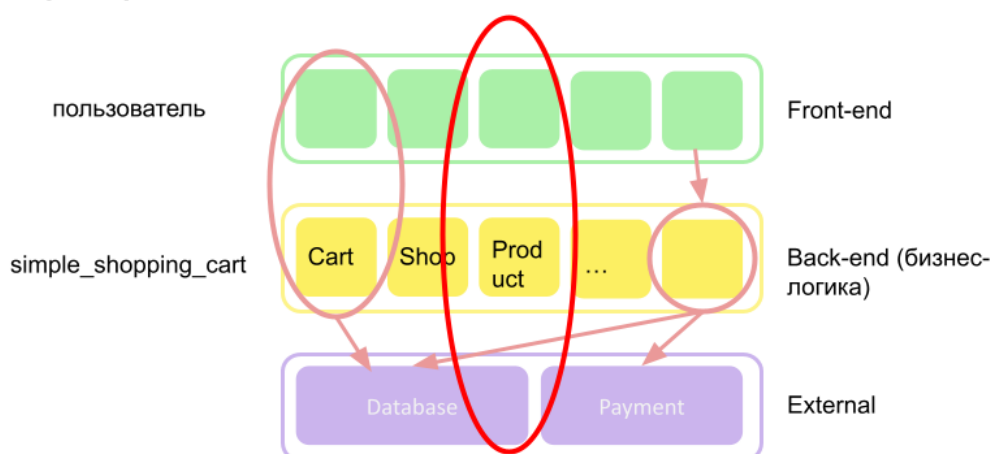
Первоначально Cucumber был разработан Ruby-сообществом, но со временем был адаптирован и для других популярных языков программирования, в том числе для java.

Теперь на примере этого же [проекта](#) рассмотрим третий тип тестирования — **сквозное тестирование**, оно же **End-to-end** или **E2E**, — это процесс тестирования, при котором происходит подробная эмуляция пользовательской среды. То есть при данном тестировании имитируют: щелчки мышью, нажатия на кнопки, заполнение форм, переходы по страницам и ссылкам, и другие поведенческие факторы.

Другие виды тестирования



### Пример. Сквозной тест



По сути, это то же тестирование двух или более компонентов, но в окружении, которое максимально повторяет окружение в котором оказывается пользователь то есть реальная работа приложения тут не должно быть никаких тестовых

двойников. Создание таких тестов — это сложная инженерная работа, поэтому таких тестов в проекте обычно немного.

## Selenium

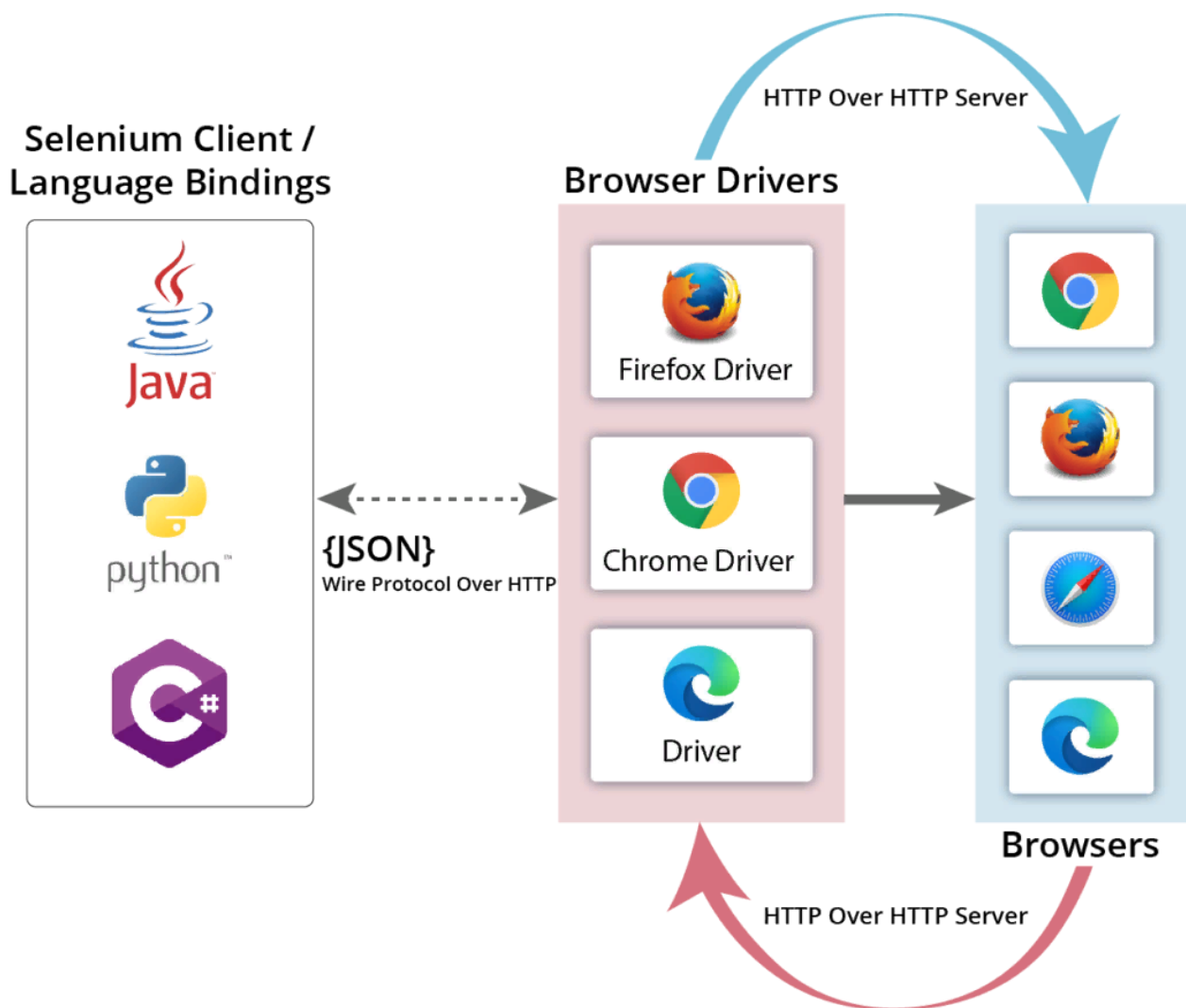
**Selenium** — платформа, используемая для автоматизации веб-приложений в браузере.

Несколько особенностей Selenium, которые способствовали его популярности:

- Open source
- Поддерживает автоматизацию на нескольких платформах, таких как Windows, Macintosh, Linux
- Поддерживает несколько браузеров, включая Chrome, Firefox, IE, Safari и др.
- Поддерживает несколько языков программирования, таких как Java, Python, Ruby, C # и др.
- Поддерживает несколько фреймворков, таких как JUnit, TestNG и т. Д.
- Легко интегрируется с инструментами CI, такими как Jenkins, Docker и др.



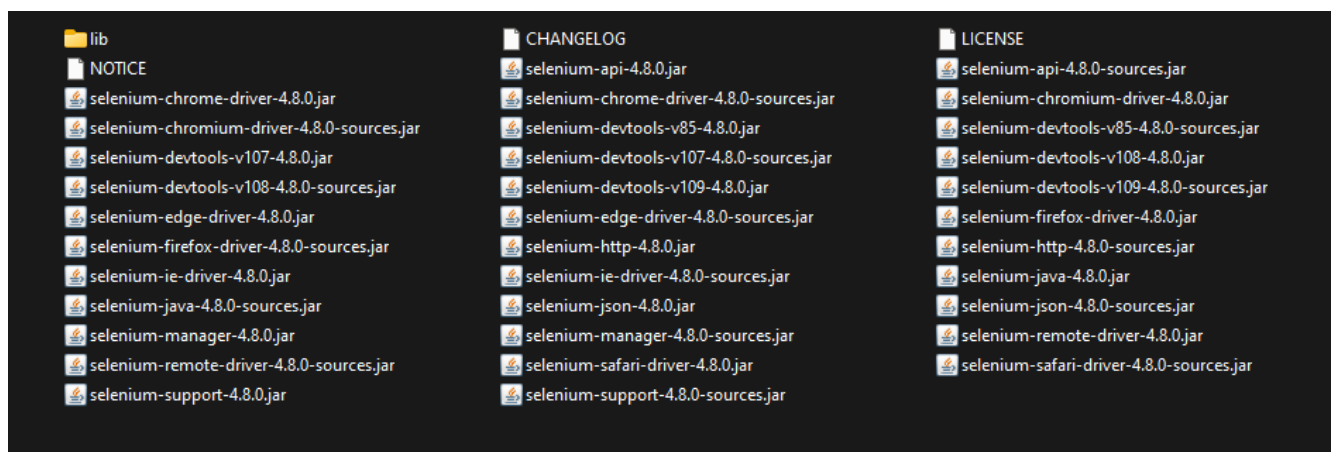
Selenium позволяет автоматизировать работу с браузерами. Это происходит путем установки драйвера, который является интерфейсом между Selenium и браузером. Драйвер посылает Selenium-команды браузеру, что позволяет Selenium управлять элементами страницы, эмулировать действия пользователя, скачивать файлы, заполнять формы и т.д.



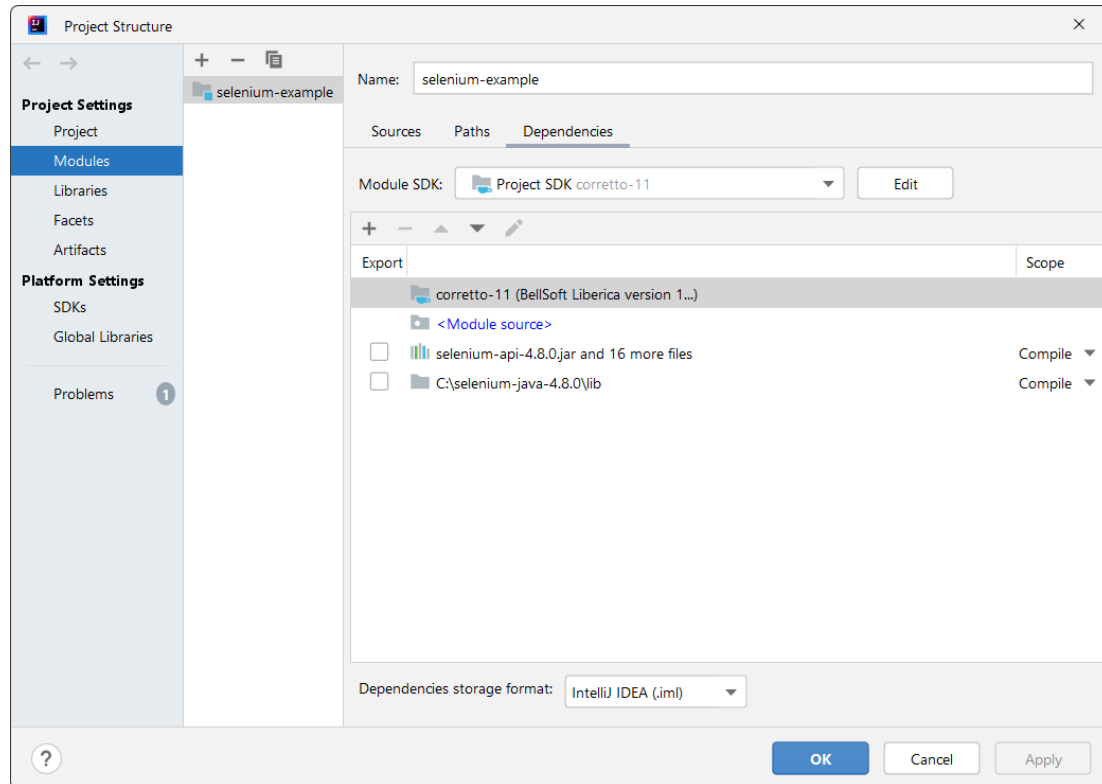
Источник: [Mavink](#)

## Установка Selenium

1. Скачиваем клиент [Downloads | Selenium](#)
2. Распаковываем архив



### 3. Добавляем зависимости к проекту



### 4. Скачиваем веб-драйвер

**Selenium WebDriver** — это программная библиотека для управления браузерами. Часто употребляется также более короткое название **WebDriver**.

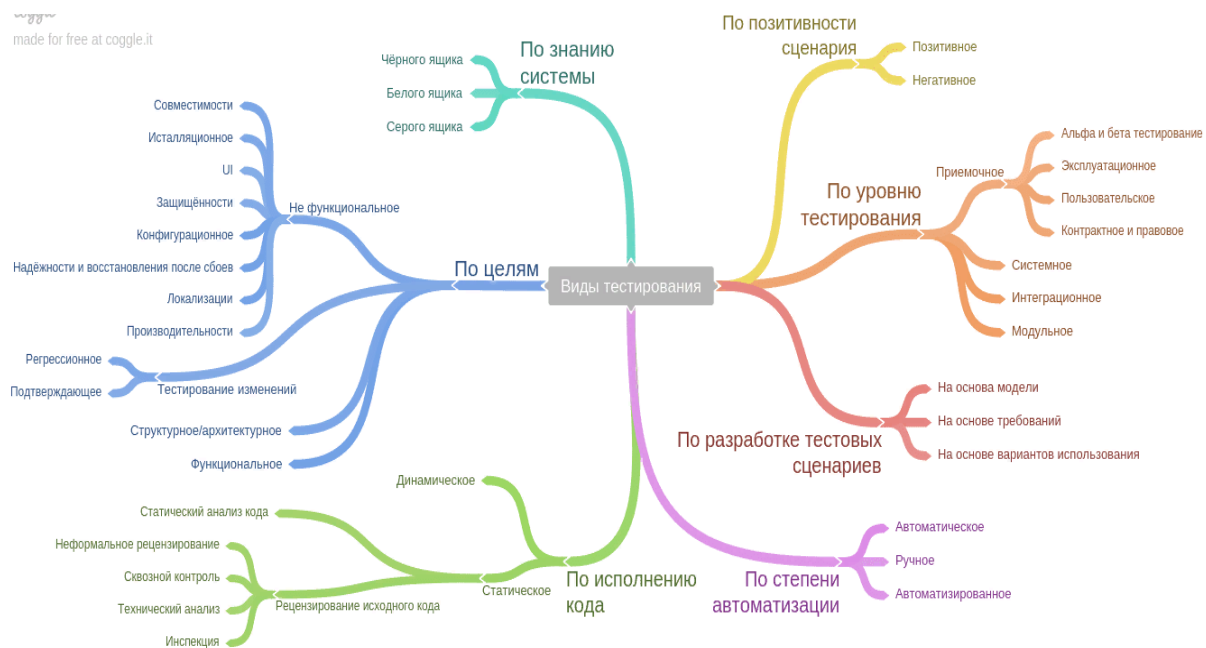
💡 Иногда говорят, что это «драйвер браузера», но на самом деле это целое семейство драйверов для различных браузеров, а также набор клиентских библиотек на разных языках, позволяющих работать с этими драйверами.

### 5. Подключаем драйвер и пробуем запустить браузер

```
public class Main {
    public static void main(String[] args) {
        System.setProperty("webdriver.chrome.driver",
"C:/chromedriver/chromedriver.exe");
        WebDriver driver = new ChromeDriver();
        driver.get("http://www.google.com/");
        WebElement searchBox = driver.findElement(By.name("q"));
        searchBox.sendKeys("GeekBrains");
        searchBox.submit();
    }
}
```

# Другие виды тестирования

made for free at [coggle.it](https://coggle.it)



Источник: [Виды тестирования](#)

В общем и целом, есть еще много разных видов тестирования и много классификаций, как видно на схеме.

Всегда сложно говорить о разных классификациях тестов. Часто нет общего мнения о том какие тесты можно назвать юнит-тестами. С интеграционными тестами ещё хуже.

Для некоторых людей интеграционное тестирование — это очень широкая деятельность, которая тестирует множество различных частей всей системы. Некоторые называют это интеграционными тестами, некоторые — компонентными, другие предпочитают термин сервисный тест. Кто-то заявит, что это вообще три совершенно разные вещи. Нет правильного или неправильного определения.

Сообщество разработчиков ПО просто не установило чётко определённых терминов в тестировании. Поэтому важно не заикливаться на двусмысленных терминах. Не имеет значения, называете вы это сквозным тестом, тестом широкого стека или функциональным тестом. Неважно, если ваши интеграционные тесты означают для вас не то, что для людей в другой компании.

Да, было бы очень хорошо, если бы наша отрасль могла четко определить термины и все бы их придерживались. К сожалению, пока это не так.

**Важно воспринимать это так:** вы просто находите термины, которые работают для вас и вашей команды. Ясно определите для себя различные типы тестов, которые

хотите написать. Согласуйте термины в своей команде и найдите консенсус относительно охвата каждого типа теста.

## Количество разных тестов в реальных проектах

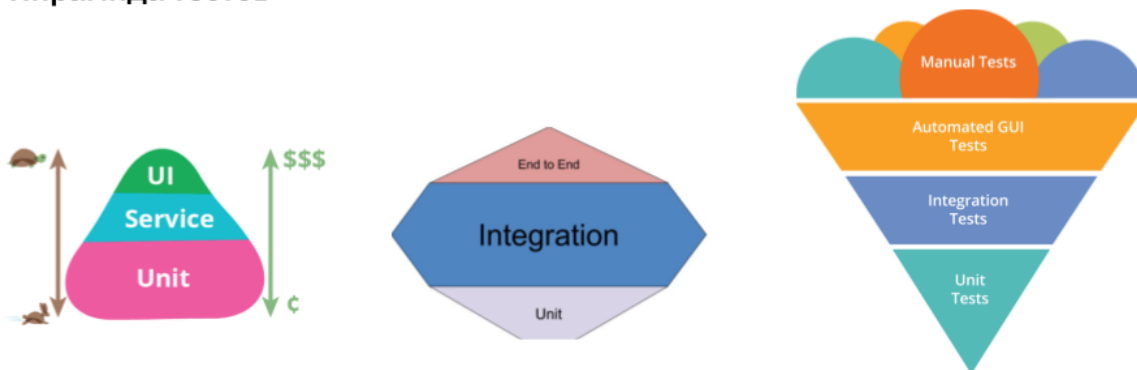
На реальных проектах преобладает разное соотношение проводимых тестов. Рассмотрим несколько вариантов развития событий.

### Пирамида тестов

Другие виды тестирования



#### Пирамида тестов



Источники: [Martin Fowler](#), [Shaun Abram](#), [System-Admins](#)

Понятие тестовой пирамиды Майка Кона (классический ее вид — первая слева на слайде выше) может казаться простой и слишком упрощенной. Тем не менее, из-за своей простоты суть тестовой пирамиды представляет хорошее эмпирическое правило, когда дело доходит до создания собственного набора тестов.

Из этой пирамиды главное запомнить два принципа:

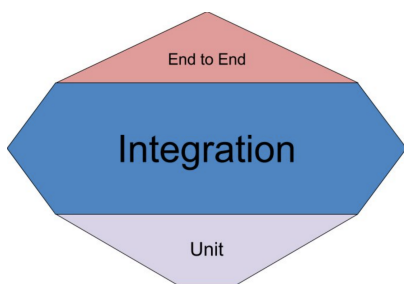
1. Писать тесты разной детализации.
2. Чем выше уровень, тем меньше тестов.

Нужно придерживаться формы пирамиды, чтобы придумать здоровый, быстрый и поддерживаемый набор тестов.

Важно помнить, что E2E тесты автоматизируются сложнее, дольше, стоят дороже, сложнее поддерживаются и трудно выполняются при регрессе. Значит таких тестов должно быть меньше



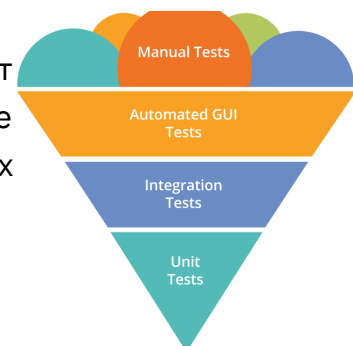
Напишите много маленьких и быстрых юнит-тестов. Напишите несколько более общих тестов и совсем мало высокоуровневых сквозных тестов, которые проверяют приложение от начала до конца. В качестве общего руководства вы можете рассмотреть возможность учета рекомендации Google 70/20/10, в которой говорится, что модульные тесты должны составлять основную часть вашего набора тестов (70%), за которыми следуют интеграционные тесты (20%) и сквозные тесты (10%)



В реальных проектах бывает следующая фигура — ромб, такое бывает так как в современных микро сервисных приложениях не так уж много бизнес-логики, все, что делают сервисы — это простые обращения к базе и передача ответа обратно, выходит что юнит-тестов в такой системе мало или они бесполезны и не выполняют полезную работу, такая схема допустима, и говорит о том, что нужно учитывать специфику проекта, и использовать тот тип тестов

которые помогут проекту развиваться

Бывает и такая фигура (вафельный рожок), к ней не стоит стремиться — это плохая практика, это когда юнит-тесты не пишутся вообще, зато много интеграционных и сквозных но основной функционал вы тестируете вручную.



## Автоматизация тестирования

Непрерывная интеграция (CI) — это практика разработки, которая подразумевает создание автоматической сборки приложения с целью быстрого обнаружения ошибок.



Мартин Фаулер (Martin Fowler) ввел термин «непрерывная интеграция» (Continuous Integration) в 2000-ом году. Целью этой методики является автоматическое сборка/тестирование проекта в различных средах выполнения, чтобы обнаруживать и устранять ошибки/конфликты

интеграции проекта, а также уменьшать расходы на последующую разработку

Обычно сборка выполняется при каждом коммите. Для этого служит специальный сервер или сервис непрерывной интеграции. Он загружает, собирает (в случае необходимости) и запускает различные тесты, линтеры, утилиты, анализирующие безопасность, актуальность зависимостей и т.д., — это зависит от усмотрения разработчика.

Для автоматизации интеграционного тестирования применяются системы непрерывной интеграции (англ. Continuous Integration System, CIS). Принцип действия таких систем состоит в следующем:

1. CIS производит мониторинг системы контроля версий;
2. При изменении исходных кодов в репозитории производится обновление локального хранилища;
3. Выполняются необходимые проверки и модульные тесты;
4. Исходные коды компилируются в готовые выполняемые модули;
5. Выполняются тесты интеграционного уровня;
6. Генерируется отчет о тестировании.

Таким образом, автоматические интеграционные тесты выполняются сразу же после внесения изменений, что позволяет обнаруживать и устранять ошибки в короткие сроки.

## Инструменты для автоматизации процессов CI

Существует достаточно много инструментов для автоматизации процессов непрерывной интеграции, тестирования и развертывания.

- **Jenkins** — один из самых популярных сервисов. Это инструмент с открытым исходным кодом, позволяющий автоматизировать процессы интеграции, тестирования и развертывания. Преимущество Jenkins перед другими сервисами — 1400 плагинов, созданных сообществом разработчиков и упрощающих работу.
- **TeamCity** — проприетарный сервис для управления сборкой и непрерывной интеграцией от российской компании JetBrains. TeamCity позволяет легко создавать образы Docker, а поддержка Jira и Bugzilla упрощает отслеживание проблем. Сервис хранит все изменения сборки и историю отказов, что



облегчает отслеживание статистики. Вдобавок разработчики могут запускать старые сборки и просматривать историю тестирования.

- **Bamboo CI** — разработка Atlassian, распространяемая по лицензии. Инструмент интегрируется с другими сервисами компании — Jira, Confluence и Clover. Bamboo CI способен выполнять параллельное тестирование с быстрой обратной связью. Это простой в освоении инструмент с удобным интерфейсом.
- **GitLab CI** — платформа с набором инструментов для работы с кодом, принадлежащая GitLab Inc. Основная часть функций доступна бесплатно, для остальных нужна покупка лицензии. Bamboo CI напрямую интегрирован с рабочим процессом GitLab и отображает всю информацию о ходе выполнения кода на единой панели инструментов.
- **Circle CI** — проприетарный сервис, разработанный одноименной компанией. Считается одним из лучших инструментов для автоматизации сборки, тестирования и развертывания. Сервис интегрируется с Bitbucket, GitHub и GitHub Enterprise, легко устраняет ошибки, быстро запускает тесты и имеет возможности для кастомизированных настроек. Он легко интегрируется с разными облачными сервисами.
- **Travis CI** — бесплатный инструмент для автоматического сбора и тестирования кода. Он позволяет разработчикам автоматически проверять их код на различных устройствах и платформах, а также уведомлять об успешных сборках.



Тестирование — один из процессов, который может быть автоматизирован в рамках CI. Надежное и тщательное автоматизированное тестирование позволяет быть уверенным в новых сборках, снижает затраты на производство и повышает качество продукта. Но даже в крупных компаниях на больших проектах оно есть не всегда. Но важно знать о возможности автоматизировать запуск тестов.

## Подведем итоги

- На этой лекции мы описали и сравнили виды тестирования: модульное, сквозное и интеграционное;
- Подробно перечислили сферы применения, достоинства и недостатки сквозного и интеграционного тестирования;

- Рассмотрели инструменты интеграционного и сквозного тестирования;
- Затронули автоматизацию тестирования и узнали про системы непрерывной интеграции.

## Домашнее задание

Для дополнительной практики попробуйте запустить браузер используя selenium, по [примеру](#) с урока.

## Контрольные вопросы

1. Что такое интеграционное тестирование Java-приложений?
2. Какие инструменты могут использоваться для интеграционного тестирования?
3. Как отличаются интеграционное и сквозное тестирование Java-приложений?

## Полезные материалы

1. [Testcontainers](#)
2. [Selenium](#)
3. [Введение в Testcontainers. Прокачиваем интеграционные тесты.](#)
4. [End to End \(e2e\) Testing React Apps With Selenium WebDriver And Node.js is Easier Than You Think](#)
5. [Автоматизация тестирования java | что такое selenium webdriver](#)
6. [Что такое CI \(Continuous Integration\)5:58](#)