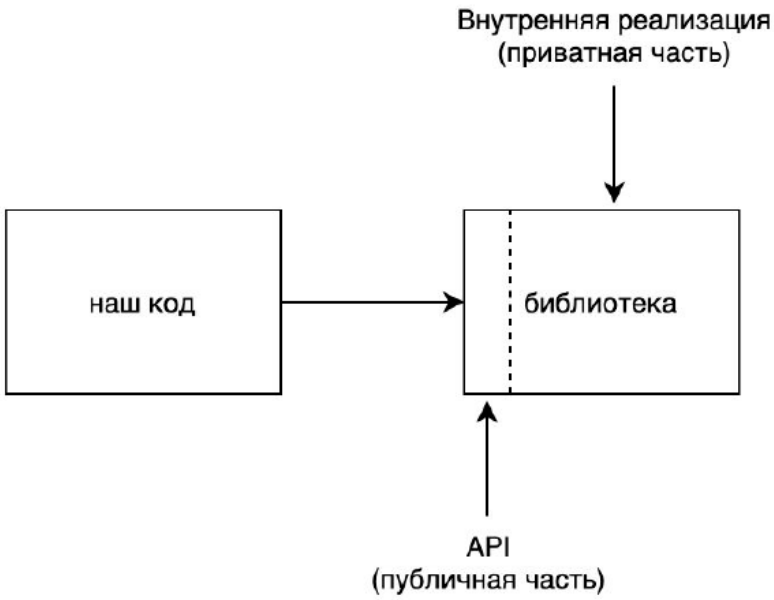
API TESTING

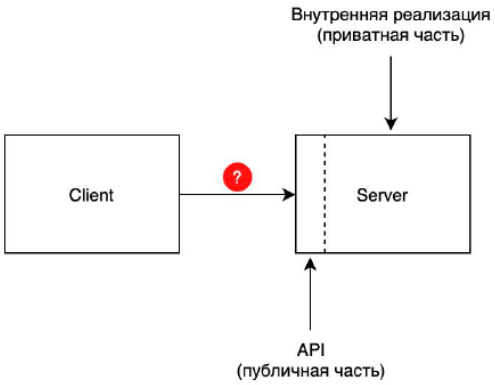
API (Application Programming Interface) — это спецификация программного взаимодействия двух систем.

Если говорить проще, то это просто определённые правила взаимодействия двух приложений (либо частей одного приложения)

> Когда мы с вами подключали библиотеки с Maven Central, то API определялось тем, какие публичные классы и методы эта библиотека представляет (т.е. что мы можем использовать).



> Начиная с сегодняшней лекции, мы выходим за рамки одного приложения (у нас больше не будет доступа к классам и методам SUT) и будем рассматривать самый популярный метод взаимодействия двух систем:

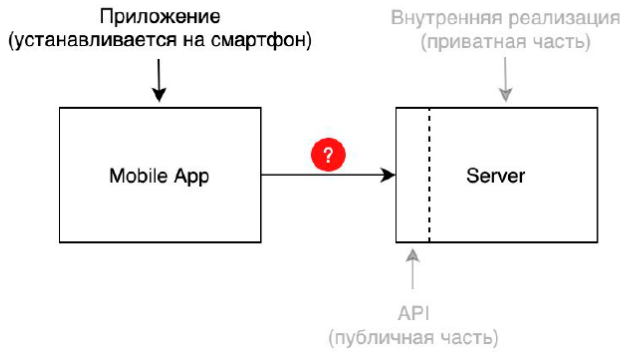


Client (клиент) — приложение, задача которого подготавливать запросыи обрабатывать ответы.

Server (сервер) — приложение, задача которого обрабатывать запросы иподготавливать ответы

В данном случае клиентом будет выступать приложение на вашем смартфоне, а сервер — какой-то сервер банка, к которому вы можете через приложение обращаться с различными запросами.

Этот же сервер отвечает за взаимодействие с хранилищами данных, в которых хранится информация о ваших счетах, картах, кредитах:



Сервер и клиент работают на разных устройствах и должны обменяться

данными в рамках обработки запроса.

Кроме того, сервер и клиент могут быть написаны на разных языках программирования.

Всё это значит, что:

1. Должен быть выбран инструмент (транспорт) для доставки информации (набора байт) от клиента к серверу.
2. Должен быть выбраны правила интерпретации этих байт (клиент исервер должны понимать друг друга).

Мы будем рассматривать достаточно распространённую схему, когда:

1. В качестве транспорта используется протокол HTTP (а именно HTTP 1.1).
2. В качестве «правил интерпретации» будет выбрано REST API с форматом данных JSON.

В сети интернет есть такое понятие как ресурс — некая информационная единица, которая

идентифицируется посредством URI (Uniform Resource Identifier).

Примеры URI:

- http://www.ietf.org/rfc/rfc2396.txt
- mailto:John.Doe@example.com
- tel:+1-816-555-1212

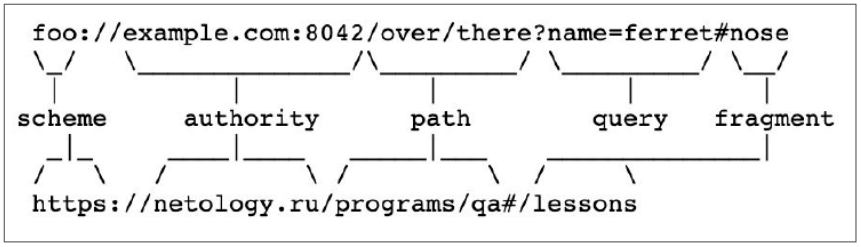
Достаточно часто вы будете слышать термин URL (Uniform Resource Locators), который является подмножеством URI.

В повседневной жизни чаще всего URI и URL используются как синонимы, что не совсем корректно.

URI состоит из трёх частей:

```
URI         = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
```

Иерархическая часть состоит из `authority` и `path`.



`authority` может включать себя логин и пароль пользователя, хост (или домен) и порт. Логин и пароль указываются достаточно редко. Порт также не указывается, если используется общепринятый (80 для http, 443 для https).

JSON (JavaScript Object Notation) — формат данных, предназначенный для передачи информации. Наряду с XML является одним из самых распространённых форматов.

Именно JSON и XML достаточно часто используют для организации обмена данными (в нашем случае между клиентом и сервером)

REST API

REST — это архитектурный подход к проектированию систем, при котором вы представляете всю систему в виде набора (чаще всего иерархических) ресурсов.

Например, есть ресурс пользователи (`users`), у каждого пользователя есть репозитории (`repos`).

Пользователи и репозитории связаны иерархическими взаимоотношениями, поэтому для списка репозиториев `/users/:username/repos`.

Для конкретного репозитория будет: `/users/:username/repos/:repo`.

Для issue репозитория: `/users/:username/repos/:repo/issues`.

Для конкретного issue: `/users/:username/repos/:repo/issues/:issue`.

Методы HTTP при этом имеют определённую смысловую нагрузку:

- GET — получить ресурс;
- POST — создать новый ресурс;
- POST/PUT/PATCH — обновить ресурс;
- DELETE — обновить ресурс.

Схема красивая, но в идеальном виде почти нереализуема (см. GitHub API).

Несмотря на это, в современном мире достаточно часто под REST имеют в виду систему разделения ресурсов по URI + HTTP-методы для смысла операции + передача JSON/XML

разработчики нам предоставили сервер в виде запускаемого JAR-файла.

Чтобы запустить его, нужно выполнить в терминале команду: java -jar app-mbank.jar

Разработчики сказали буквально следующее: "делаешь GET запрос на http://localhost:9999/api/v1/demo/accounts, в ответ получишь JSON":


```
[
  {
    "id": 1,
    "name": "Текущий счёт",
    "number": "•• 0668",
    "balance": 992821429,
    "currency": "RUB"
  },
  {
    "id": 2,
    "name": "Текущий счёт",
    "number": "•• 9192",
    "balance": 2877210,
    "currency": "USD"
  },
  {
    "id": 3,
    "name": "Текущий зарплатный счёт",
    "number": "•• 5257",
    "balance": 1204044,
    "currency": "RUB"
  }
]
```


Вы уже знакомы с Postman: инструмент, который позволяет вам делать запросы и даже производить тестирование с элементами автоматизации.

Но нам хотелось бы делать всё на Java. Для этого у нас есть REST-assured:

```
dependencies {
  testImplementation 'org.junit.jupiter:junit-jupiter:5.6.1'
  testImplementation 'io.rest-assured:rest-assured:4.3.0'
}

test {
  useJUnitPlatform()
}
```

Важно: всегда читайте документацию (зависимости должны идти именно в этом порядке).

Достаточно часто автоматизаторы сначала проверяют запросы в Postman (либо аналоге), а затем переносят всё в код.

```java
@Test
void shouldReturnDemoAccounts() {
  // Подход: Given – When – Then
  // Предусловия
  given()
      .baseUri("http://localhost:9999/api/v1")
  // Выполняемые действия
  .when()
      // get метод запроса GET
      // URI относительно baseUri
      .get("/demo/accounts")
  // Проверки
  .then()
      .statusCode(200);
}
```

Всё, что делает наш тест – удостоверяется, что код ответа равен 200 (OK).

В зависимости от постановки задачи мы можем тестировать:

- возвращаемые заголовки

```java
@Test
void shouldReturnDemoAccounts() {
  // Given – When – Then
  // Предусловия
  given()
      .baseUri("http://localhost:9999/api/v1")
  // Выполняемые действия
  .when()
      .get("/demo/accounts")
  // Проверки
  .then()
      .statusCode(200)
      .header("Content-Type", "application/json; charset=UTF-8")
      // специализированные проверки – лучше
      .contentType(ContentType.JSON)
  ;
}
```

- тело ответа (если оно есть)

```java
@Test
void shouldReturnDemoAccounts() {
  ... // аналогично предыдущему коду
      .body("", hasSize(3))
      .body("[0].id", equalTo(1))
      .body("[0].currency", equalTo("RUB"))
      .body("[0].balance", greaterThanOrEqualTo(0))
  ;
}
```

REST-assured использует специальный синтаксис для доступа к элементам внутри JSON:

- `""` – корневой элемент;
- `[0]` – доступ к элементу массива по индексу;
- `.id` – доступ к свойству объекта.

Итого, получается что `[0].id` – это доступ к полю `id` первого элемента массива.

Кроме того, REST-assured предлагает использовать нам библиотеку Hamcrest, которая содержит набор удобных `matcher`'ов.

На самом деле, под «капотом» REST-assured использует Groovy, а именно GPath для обработки этих выражений и если вы будете плотно работать с REST-assured, то хорошо бы его выучить:

```
@Test
void shouldReturnDemoAccounts() {
  ... // аналогично предыдущему коду
  .then()
      .statusCode(200)
      .contentType(ContentType.JSON)
      .body("every{ it.balance >= 0 }", is(true)) // все балансы неотрицательные
  ;
}
```

Не всегда нам нужно проверить именно сами данные, иногда нам нужно проверить соответствие их определённой структуре и ограничениям (схеме).

Для JSON есть специальный проект JSON Schema, который позволяет проверять документ на соответствие схеме.

Как же запускать сервер для тестов?

Вариантов достаточно много, мы рассмотрим несколько ключевых:

1. Делегировать эту задачу кому-то другому (SUT уже должен быть запущен).

Достаточно удобный для нас вариант, нам нужно знать только URL SUT,на который нужно посылать запросы.

Тесты пишутся и запускаются как обычно.

Нам придётся столкнуться с рядом проблем, которые мы будем

обсуждать на протяжении всего курса:

- периодическое падение тестов из-за недоступности сервера (например, из-за проблем сети между CI и сервером);
- сложность «управления» SUT (например, установки начального состояния, перезапуска в нужной конфигурации и т.д.);
- зависимость тестов от текущего SUT или его текущей версии (команда, отвечающая за тестовый сервер, может решить перезапустить его во время наших тестов или обновить).

Доходит до того, что иногда сервер, на котором развёрнута SUT, могут вообще временно «изъять под другие задачи».😈

К сожалению, это достаточно часто встречается на практике и приводит к необходимости разбирать FAIL'ы.

«Усталость» от разбора таких «интеграционных» проблем тестов приводит к выработке вредной привычки:

если тесты падают, то, возможно, это из-за интеграции, попробуем их просто перезапустить.

Это очень плохо! Подобный подход приводит к тому, что доверие к тестам пропадает => тесты перестают

использоваться (их результат не учитывается) => смысла в тестах нет.

Конечно же, мы описываем то, как должно быть, и в реальной практике вы достаточно часто встретитесь

с подходом «перезапустим тесты».

2. Запускать сервер средствами CI.

И CI, и Gradle, и Java позволяют нам запускать помимо тестов другие приложения (речь о процессах*). Поскольку CI предоставляет возможность запуска произвольных команд, то мы вполне может этим воспользоваться исходя из того, какую командную оболочку (CMD, PowerShell, Bash и т.д.) и операционную систему предоставляет CI.

В Gradle мы можем использовать Exec.

В Java для запуска команд существуют инструменты Runtime.exec() и ProcessBuilder.

Достаточно удобный для нас вариант: мы сами запускаем SUT; проблем сети, перезагрузки сервера с SUT или обновления SUT другой командой не будет.

Эти варианты также обладают рядом проблем:

- *SUT должна быть спроектирована для подобного запуска (установка и запуск SUT могут*

представлять из себя комплексную процедуру);

- *зависимости SUT (SUT может требовать баз данных, систем очередей или интеграции с реальными*

системами);

- *у вас должен быть актуальная версия SUT (тестировщики и разработчики достаточно часто работают*

в разных подразделениях);

- *требования SUT должны быть соизмеримы с техническими характеристиками CI.*

3. Запускать сервер из тестов (Gradle или Java).