Мок сервисы. Когда интеграции толком нет, а тестировать надо уже сейчас

Многие приложения и сервисы редко работают в изоляции, чаще всего они интегрируются, могут быть расположены как в локалке, так и на удаленных серверах. Когда встаёт вопрос тестирования интеграции в рамках общего сценария, часто можно столкнуться с различными проблемами.

Давайте предположим, что мы ведем разработку функционала, который обращается к удаленному сервису, получает от него данные и далее обрабатывает ответ. Не важно какие именно данные мы отправляем(файлы, структурированные данные или что-то другое), важно то, что эти данные влияют на работу алгоритмов и всей нашей системы в целом. Внешний удаленный сервис работает крайне нестабильно, глючит, выдает ошибки и прочее, что замедляет процесс тестирования нашей системы т.к. мы не можем нормально протестить интеграцию. Согласитесь, подобные условия встречаются часто, в таких случаях мы оставляем вопрос тестирования на потом, переключаясь на другие задачи. А что если надо протестировать уже сейчас?  
  
Предположим сценарий.

Обращение к внешнему веб-серверу, который разрабатывается не нами и который работает нестабильно, постоянно падает, тупит и вообще черепаха, мы отказываемся работать в таких условиях и вообще непорядок, надо что-то делать, потому что сроки горят.   
  
В таком случае, чтобы наши тесты работали стабильно и не зависели от сторонних систем, нам нужна некая копия внешнего сервиса, которая «ты во время тестов не ходи на удаленный сервер, ты ходи на наш»  
  
Для этой задачи можно поднять упрощенный вариант нашего удаленного сервиса, который будет называться мок-сервисом.

Основные причины внедрения моков:  
1. Тесты замедляются из-за медленной работы внешних сервисов

2. тесты нестабильны т.к. внешний сервис работает нестабильно

3. не полное покрытие тестами.

4. недоступность внешнего сервиса из-за безопасности. (ведем разработку в одном контуре, а внешний сервис в другом)

5. риск использования реальных данных и, собственно, их утечки или порчи.

Для того, чтобы стандартизировать разделение мок-сервисов по тому, как они работают, есть условное разделение по сложности обработки входящих запросов и тому, что возвращается в ответе.

Типы моков:  
1.Dummy. пустые объекты, которые заполняются только для параметров методов (нужен, чтобы проверить только то, что API вообще есть и что-то отправляется)

2. Fake реальные объекты, но в таком виде, которые не подходят для рабочей ситуации (оптимальный вариант, но невозможно проверить ответ и влияние ответа на нашу систему. Используются для того, чтобы проверить структуру запроса и подготовить структура ответа)

3. Stub заранее заготовленные ответы на вызовы во время теста, не отвечают на другие вызовы, которые требуются в тесте (что-то вроде хардкода, позволяет приблизиться к реальным условиям)

4. Mock объекты, которые полностью заменяют реальный объект в условиях теста, позволяют проверить вызовы своих методов. (наиболее близкий вариант к реальным, но трудозатратный)

Соответственно, все эти варианты как раз решают нашу задачу по тестированию интеграции, однако тут есть 2 вполне себе логичные проблемы:

Проблема 1. Моки надо писать, описывать логику и все вот это вот. Необходимо закладывать время на написание заглушки,

Проблема 2. Сам сервис может поменяться, мок придется переписывать.

Решение этих двух проблем простое. Надо сделать набор смоук-тестов для каждого эндпоинта в API и ограничиться простым Stub, который быстро пишется и его несложно поддерживать. В результате смоук тесты позволят следить за актуальной версией того, что отправляется от нашей системы(Спецификация API), а Stub позволит все это безобразие проверить.

Итого, формируем первое требование к нашему мок-сервису.   
1. Это должен быть простой и легко поддерживаемый Stub, за которым смогут следить не только разработчики, но и тестировщики.  
  
А теперь важный вопрос. А где его развернуть, локально на той же тачке, где и наши сервисы, или же удаленно на отдельной тачке?  
  
Поскольку у нас есть такая прекрасная штука как контейнеризация, то можно просто развернуть контейнер с этим сервисом и запустить его.  
  
Следовательно, второе требование.   
Мок должен легко запускаться внутри контейнера.

Ну и поскольку мы следуем определенным стандартам внутри компании, то для этого будем использовать Java и готовые фреймы, чтобы не изобретать велосипед.  
  
Фреймов довольно много, таких как   
RestMock  
Okhttpmockwebserver  
WireMock  
Mockito

PowerMock и так далее

Когда гуглил мок сервисы, то чаще всего глаза мазолил WireMock, на нем и остановимся, объективных причин не использовать другие фреймы нет, на вкус и цвет все фломастеры разные, в моем случае это просто вопрос привычки☺

Переходим к коду.   
\*занавес открывается, появляется идея с заранее написанным кодом\*

Поскольку мы рассматриваем простой пример, то будем с ним работать по принципу лабораторных работ в университете, все пишем в одном классе на одной странице.

Конфиг  
  
Показать как конфигурируется ip адрес, на котором будет работать мок  
Показать порт. Сделать оговорку, что этот порт надо будет предварительно открыть через консоль линукса командой chmod

Диспатчер.  
  
Диспатчер это набор условий для работы с принятым запросом. В этой штуке прописывается все, что необходимо для того, чтобы обработать запрос и вернуть ответ.  
  
(показать где конфигурируется http метод, параметры, тело запроса)  
(показать где конфигурируется код ответа и тело ответа на примере запроса Get\_employee)

(Открыть постман, поотправлять запросы валидные и невалидные и показать что будет)

Далее усложняем задачу. Нам надо спарсить параметры, которые передаются в запросе и вернуть их в теле ответа. Для этого обращаемся к HandleBars Helpers. Прописываем request.query.\*название параметра\* и получаем наш параметр в виде String.

Усложняем задачу. Теперь нам надо спарсить body запроса в виде JSON и забрать оттуда определенные параметры. Для этой задачки используем jsonPath и используем тот же request.body. Ну и вдогонку можем добавить какое-нибудь рандомное значение. В данном случае используем генератор UUID.

Теперь еще сложнее задачка. Мы отправляем запрос, а в ответе нам должен вернуться файлик.

Тут остановимся и рассмотрим эту функцию подробнее.

У WireMock есть своя прикольная фишка, которая заключается в том, что у него есть API спецификация, с помощью которой можно сильно упростить себе работу.

<https://github.com/wiremock/wiremock/blob/2.19.0/src/main/java/com/github/tomakehurst/wiremock/admin/AdminRoutes.java#L77>

Согласно этой спецификации, используемые файлы лежат в src.test.resources.\_\_files

Поэтому можно положить его заранее, а можно будет отправить потом на уже запущенный мок сервис через специальный эндпоинт WireMock, который указан в этой спеке. Работа с файлами из других папок не ведется, такова политика безопасности, чтобы мы не могли залезть случайно куда-нибудь и что-нибудь сделать.

Далее усложняем себе задачку. Представим, что у нас адрес один, но он выдает разные данные в зависимости от используемого хедера. Тут просто перед Will return прописываем withHeader и заполняем нужными значениями.

А еще можно проверить базовую авторизацию. Добавим withBasicAuth и готово!

Итог.   
В целом, мок как инструмент штука полезная, их хорошо использовать на больших проектах, не важно, мобилки это или нет. Но и не без минусов. Простые заглушки пишутся быстро, на примере данного проекта можно разработать простой мок сервис, который сможет поддерживать даже человек, который не знаком с ООП. Однако могут появиться проблемы со сложными моками. В данном случае проще обратиться к разработчикам. Разработке в то же время проще обвесить интеграцию тестами, а не писать заглушку, которую потом еще поддерживать надо будет. Поэтому мне кажется, что правильней будет, если моками занимаются именно тестировщики.  
На этом у меня все, готов ответить на вопросы ☺

\*ссылка на личку в ТГ\*  
\*ссылка на гит КРОК\*