опыт применения автотестов в ВТБ

**8 мин**

**8.6K**

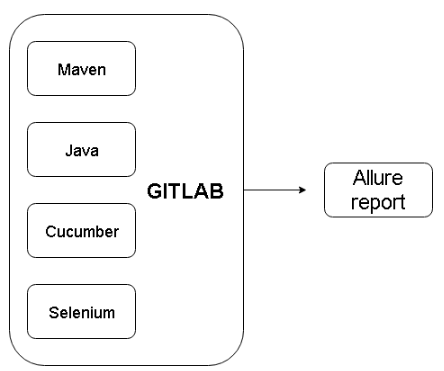
[Блог компании ВТБ](https://habr.com/ru/companies/vtb/articles/)[Тестирование IT-систем\*](https://habr.com/ru/hubs/it_testing/)[Java\*](https://habr.com/ru/hubs/java/)[IT-инфраструктура\*](https://habr.com/ru/hubs/it-infrastructure/)[Финансы в IT](https://habr.com/ru/hubs/finance/)

Наше подразделение создает полностью автоматические пайплайны для вывода новых версий приложений в прод-среду. Разумеется, для этого требуются автоматизированные функциональные тесты. Под катом — история о том, как, начав с тестирования в один поток на локальной машине, мы дошли до многопоточного запуска автотестов на Selenoid в пайплайне сборки с Allure-отчетом на GitLab pages и в итоге получили крутой инструмент для автоматизации, который смогут использовать будущие команды.

С чего мы начинали

Чтобы реализовать автотесты и встроить их в пайплайн, нам требовался фреймворк для автоматизации, который можно гибко менять под наши потребности. В идеале хотелось получить единый стандарт для движка автотестирования, приспособленный под встраивание автотестов в пайплайн. Для реализации мы выбрали следующие технологии:

* Java,
* Maven,
* Selenium,
* Cucumber+JUNIT 4,
* Allure,
* GitLab.

  
  
Почему именно такой набор? Java — один из самых популярных языков для автотестов, к тому же им владеют все члены команды. Selenium — очевидное решение. Cucumber же, помимо прочего, должен был повысить доверие к результатам автотестов со стороны подразделений, занимающихся ручным тестированием.

Однопоточные тесты

Чтобы не изобретать велосипед, за основу фреймворка мы взяли наработки с различных репозиториев на GitHub и адаптировали их под себя. Создали репозиторий для главной библиотеки с ядром фреймворка автотестов и репозиторий с Gold-примером реализации автотестов на нашем ядре. Каждая команда должна была брать Gold-образ и разрабатывать в нем тесты, адаптируя под свой проект. Развернули в банке GitLab-CI, на котором настроили:

* ежедневные запуски всех написанных автотестов по каждому проекту;
* запуски в пайплайне сборки.

Сначала тестов было мало, и они шли в один поток. Однопоточный запуск на Windows-раннере GitLab нас вполне устраивал: тесты очень незначительно нагружали тестовый стенд и почти не утилизировали ресурсы.

Со временем автотестов становилось все больше, и мы задумались над их параллельным запуском, когда полный прогон стал занимать около трех часов. Появились и другие проблемы:

* мы не могли убедиться в том, что тесты стабильны;
* тесты, которые проходили по несколько прогонов подряд на локальной машине, иногда падали в CI.

Пример настройки автотестов:

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.20</version>

<configuration>

<skipTests>${skipTests}</skipTests>

<testFailureIgnore>**false**</testFailureIgnore>

<argLine>

-javaagent:"${settings.localRepository}/org/aspectj/aspectjweaver/${aspectj.version}/aspectjweaver-${aspectj.version}.jar"

-Dcucumber.options="--tags ${TAGS} --plugin io.qameta.allure.cucumber2jvm.AllureCucumber2Jvm --plugin pretty"

</argLine>

</configuration>

    <dependencies>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>${aspectj.version}</version>

</dependency>

</dependencies>

</plugin>

<plugin>

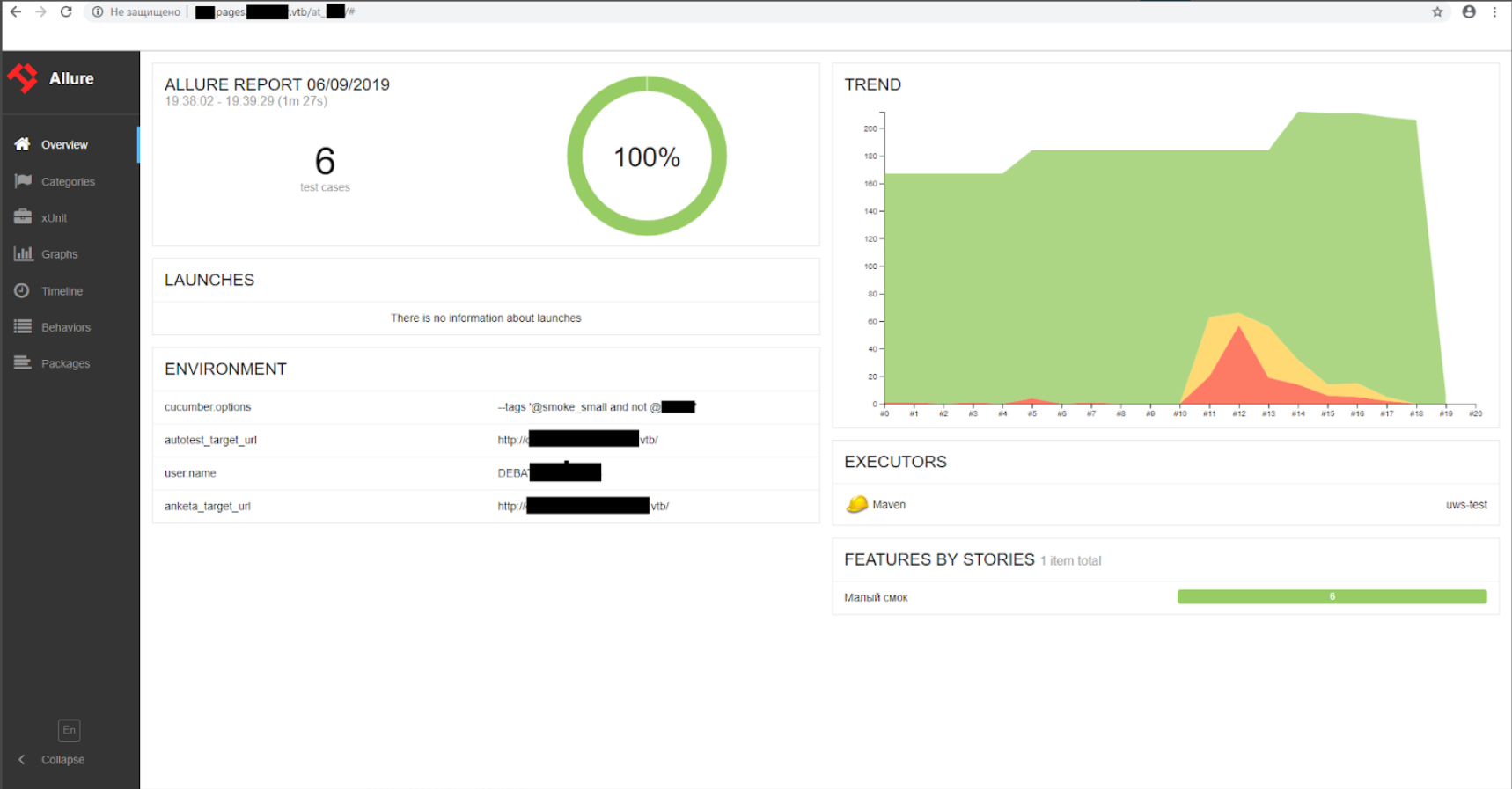
<groupId>io.qameta.allure</groupId>

<artifactId>allure-maven</artifactId>

<version>2.9</version>

</plugin>

</plugins>

   
*Пример Allure-отчета*  
  
   
*Нагрузка на раннер во время тестов (8 ядер, 8 ГБ ОЗУ, 1 поток)*  
   
Плюсы однопоточных тестов:

* легко настроить и запустить;
* запуски в CI практически не отличаются от локальных запусков;
* тесты не аффектят друг друга;
* минимальные требования к ресурсам раннера.

Минусы однопоточных тестов:

* очень долго выполняются;
* долгая стабилизация тестов;
* неэффективное использование ресурсов раннера, крайне низкая утилизация.

Тесты на форках JVM

Поскольку при реализации базового фреймворка мы не позаботились о thread-safe коде, самым очевидным способом параллельного запуска стал[cucumber-jvm-parallel-plugin](https://github.com/temyers/cucumber-jvm-parallel-plugin) для Maven. Плагин легко настроить, но для корректной параллельной работы автотесты нужно запускать в отдельных браузерах. Делать нечего, пришлось использовать Selenoid.

Selenoid-сервер подняли на машине с 32 ядрами и 24 ГБ ОЗУ. Лимит установили в 48 браузеров — 1,5 потока на ядро и порядка 400 МБ ОЗУ. В итоге время тестов сократилось с трех часов до 40 минут. Ускорение прогонов помогло решить проблему стабилизации: теперь мы могли быстро прогонять новые автотесты 20–30 раз, пока не убедимся, что они выполняются стабильно.  
Первым недостатком решения стала высокая утилизация ресурсов раннеров при небольшом количестве параллельных потоков: на 4 ядрах и 8 ГБ ОЗУ тесты работали стабильно не более чем в 6 потоков. Второй минус: плагин генерирует классы-раннеры для каждого сценария, сколько бы их ни запускалось.

**Важно!**Не прокидывайте переменную с тегами в ***argLine***, например, так:

<argLine>-Dcucumber.options="--tags ${TAGS} --plugin io.qameta.allure.cucumber2jvm.AllureCucumber2Jvm --plugin pretty"</argLine>

…

Mvn –DTAGS="@smoke"

Если передавать тег таким образом, плагин будет генерировать раннеры для всех тестов, то есть пытаться запустить все тесты, пропуская их сразу после запуска и создавая при этом множество форков JVM.

Правильно переменную с тегом прокидывать в ***tags*** в настройках плагина, см. пример ниже. В других проверенных нами способах возникают проблемы с подключением плагина Allure.  
  
Пример времени прогона 6 коротких тестов с неправильной настройкой:

[INFO] Total time: 03:17 min

Пример времени прогона тестов, если напрямую передавать тег в **mvn … –Dcucumber.options**:

[INFO] Total time: 44.467 s

Пример настройки автотестов:

<profiles>

<profile>

<id>parallel</id>

<build>

<plugins>

<plugin>

<groupId>com.github.temyers</groupId>

<artifactId>cucumber-jvm-parallel-plugin</artifactId>

<version>5.0.0</version>

<executions>

<execution>

<id>generateRunners</id>

<phase>generate-test-sources</phase>

<goals>

<goal>generateRunners</goal>

</goals>

<configuration>

            <tags>

<tag>${TAGS}</tag>

</tags>

<glue>

<**package**>stepdefs</**package**>

</glue>

</configuration>

              </execution>

</executions>

        </plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

        <version>2.21.0</version>

<configuration>

<forkCount>12</forkCount>

<reuseForks>**false**</reuseForks>

<includes>\*\*/\*IT.class</includes>

<testFailureIgnore>false</testFailureIgnore>

<!--suppress UnresolvedMavenProperty -->

<argLine>

-javaagent:"${settings.localRepository}/org/aspectj/aspectjweaver/${aspectj.version}/aspectjweaver-${aspectj.version}.jar" -Dcucumber.options="--plugin io.qameta.allure.cucumber2jvm.AllureCucumber2Jvm TagPFAllureReporter --plugin pretty"

</argLine>

</configuration>

<dependencies>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>${aspectj.version}</version>

  </dependency>

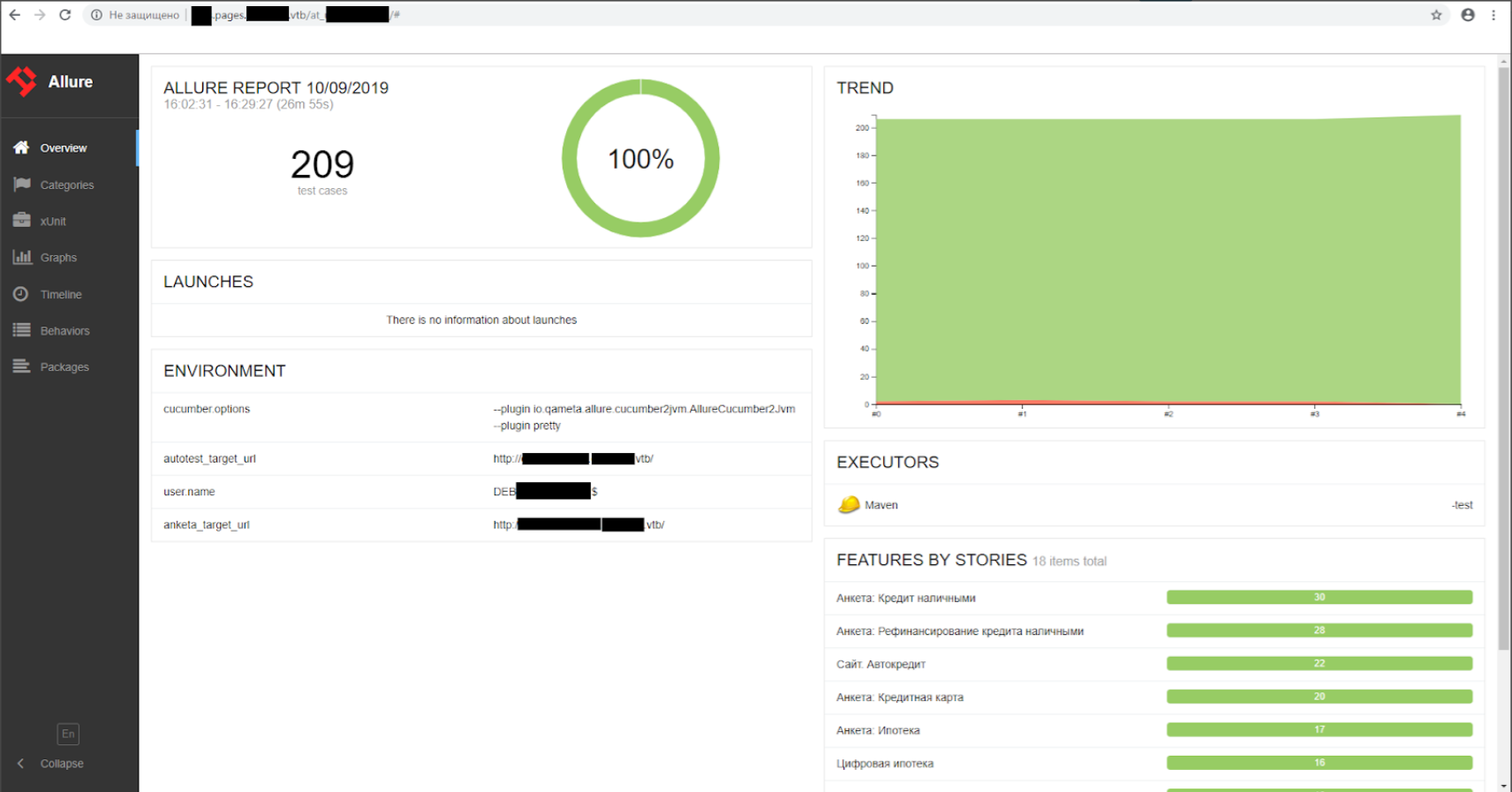
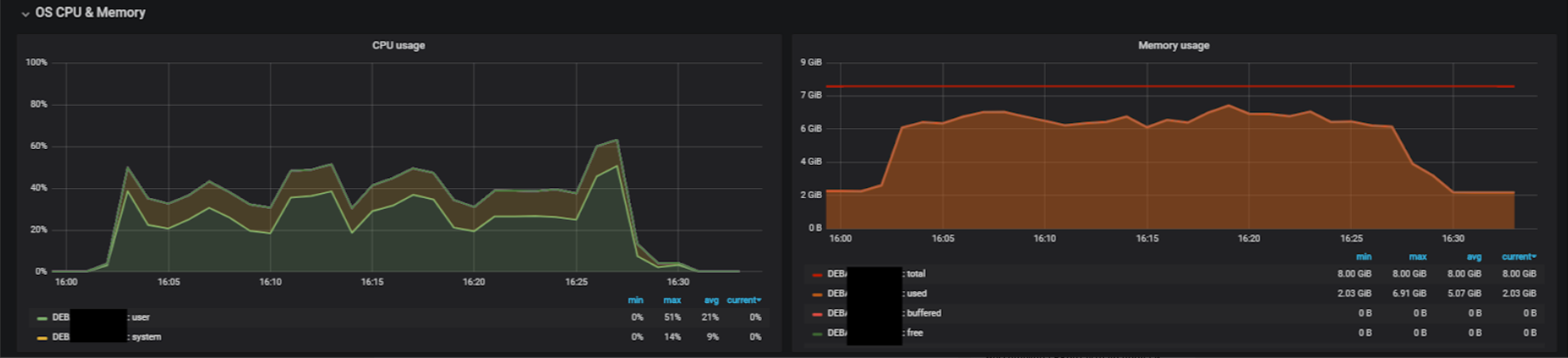
</dependencies>

  </plugin>

</plugins>

</build>

</profile>

  
*Пример Allure-отчета (самый нестабильный тест, 4 рерана)*  
  
*Нагрузка раннера во время тестов (8 ядер, 8 ГБ ОЗУ, 12 потоков)*  
   
Плюсы:

* простая настройка — нужно лишь добавить плагин;
* возможность одновременного выполнения большого количества тестов;
* ускорение стабилизации тестов благодаря п.1.

Минусы:

* требуется несколько ОС / контейнеров;
* высокое потребление ресурсов на каждый форк;
* плагин устарел и больше не поддерживается.

Как победить нестабильность

Тестовые стенды не идеальны, как и сами автотесты. Неудивительно, что у нас появилось некоторое количество flacky-тестов. На помощь пришел[maven surefire plugin](https://www.apache-maven.ru/plugins/maven-surefire-plugin.html), который из коробки поддерживает перезапуск упавших тестов. Нужно обновить версию плагина минимум до 2.21 и написать одну строчку с количеством перезапусков в pom-файле или передать в качестве аргумента для Maven.

Пример настройки автотестов:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

      <artifactId>maven-surefire-plugin</artifactId>

<version>2.21.0</version>

<configuration>

….

<rerunFailingTestsCount>2</rerunFailingTestsCount>

….

</configuration>

</plugin>

Либо при запуске: ***mvn … -Dsurefire.rerunFailingTestsCount=2 …***  
Как вариант – задать опции Maven для скрипта PowerShell (PS1):

Set-Item Env:MAVEN\_OPTS "-Dfile.encoding=UTF-8 -Dsurefire.rerunFailingTestsCount=2"

Плюсы:

* не нужно тратить время на анализ нестабильного теста, когда он падает;
* можно сгладить проблемы стабильности тестового стенда.

Минусы:

* можно пропустить плавающие дефекты;
* время прогона увеличивается.

Параллельные тесты с библиотекой Cucumber 4

Количество тестов росло с каждым днем. Мы снова задумались об ускорении прогонов. Кроме того, хотелось встроить в пайплайн сборки приложения как можно больше тестов. Критическим фактором стала слишком долгая генерация раннеров при запуске в параллель с помощью Maven-плагина.  
  
На тот момент уже вышел Cucumber 4, поэтому мы решили переписать ядро под эту версию. В release notes нам обещали параллельный запуск на уровне тредов. Теоретически это должно было:

* значительно ускорить прогон автотестов за счет увеличения количества потоков;
* исключить потерю времени на генерацию раннеров для каждого автотеста.

Оптимизировать фреймворк для многопоточных автотестов оказалось не так сложно. Cucumber 4 прогоняет каждый отдельный тест в выделенном потоке от начала и до конца, поэтому некоторые общие static-вещи были просто преобразованы в ThreadLocal-переменные.   
Главное при конвертации средствами рефакторинга Idea — проверить места, в которых происходило сравнение переменной (например, проверка на null). Кроме того, нужно вынести Allure-плагин в аннотации класса Junit Runner.

Пример настройки автотестов:

<profile>

<id>parallel</id>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

          <artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M3</version>

        <configuration>

<useFile>**false**</useFile>

<testFailureIgnore>**false**</testFailureIgnore>

        <parallel>methods</parallel>

<threadCount>6</threadCount>

<perCoreThreadCount>**true**</perCoreThreadCount>

<argLine>

-javaagent:"${settings.localRepository}/org/aspectj/aspectjweaver/${aspectj.version}/aspectjweaver-${aspectj.version}.jar"

</argLine>

</configuration>

<dependencies>

<dependency>

<groupId>org.aspectj</groupId>

                <artifactId>aspectjweaver</artifactId>

<version>${aspectj.version}</version>

</dependency>

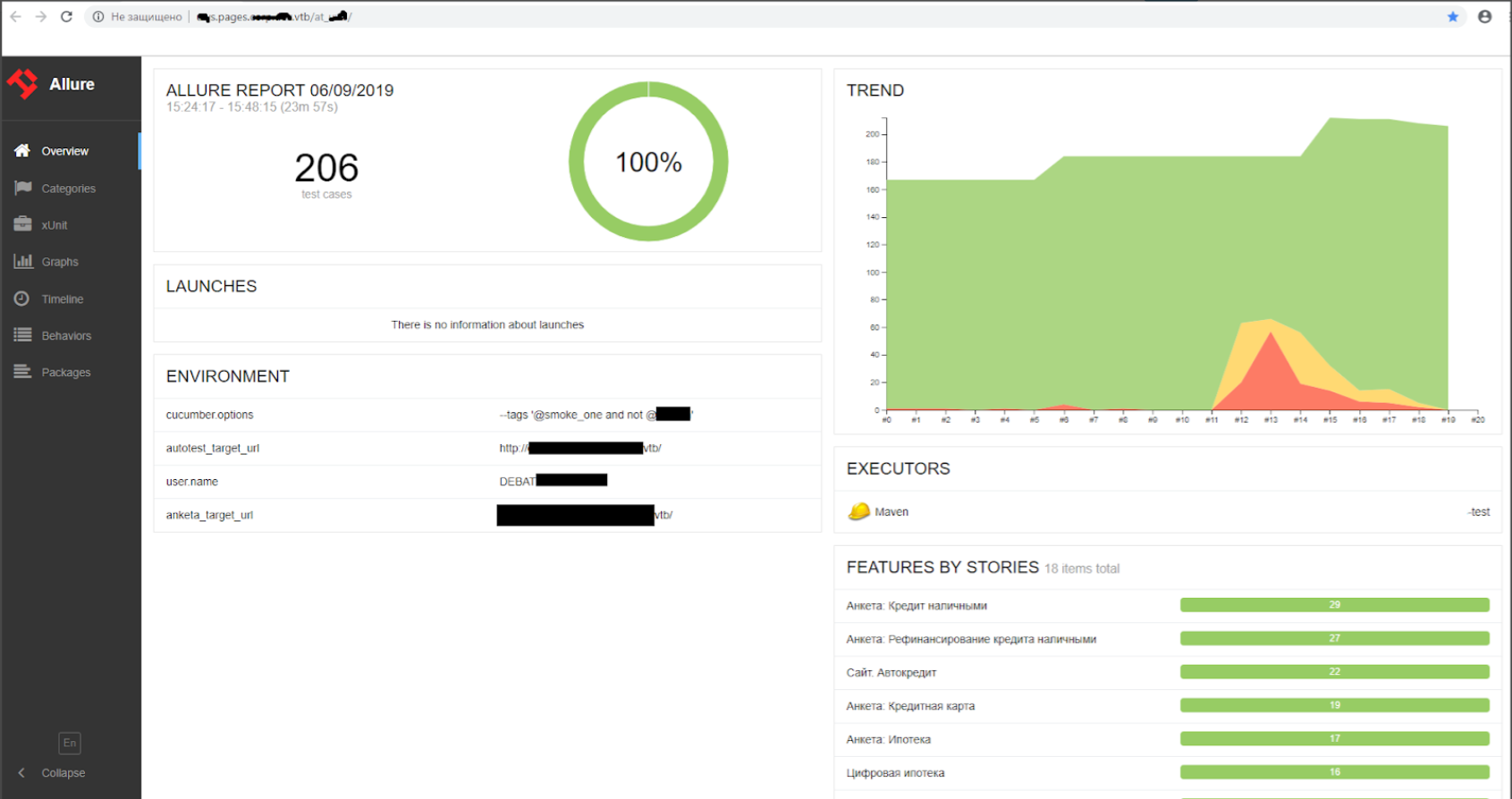
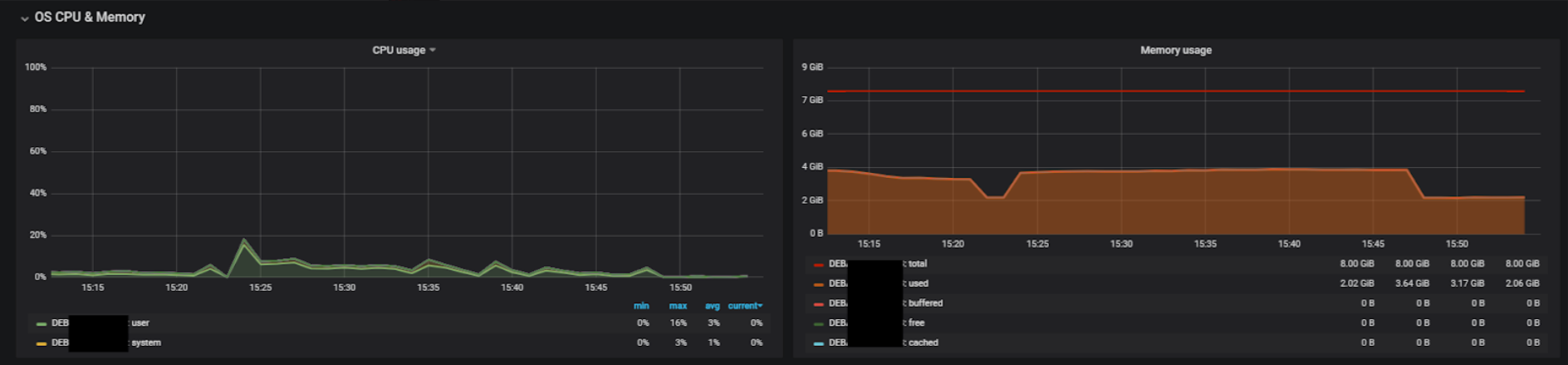
</dependencies>

</plugin>

</plugins>

</build>

</profile>

  
*Пример Allure-отчета (самый нестабильный тест, 5 реранов)*  
  
*Нагрузка раннера во время тестов (8 ядер, 8 ГБ ОЗУ, 24 потока)*  
  
Плюсы:

* низкое потребление ресурсов;
* нативная поддержка от Cucumber — не нужны дополнительные инструменты;
* возможность запуска более 6 потоков на ядро процессора.

Минусы:

* нужно следить за тем, чтобы код поддерживал многопоточное выполнение;
* увеличивается порог вхождения.

Allure-отчеты в GitLab pages

После внедрения многопоточного запуска мы стали тратить на анализ отчетов гораздо больше времени. На тот момент нам приходилось выкладывать каждый отчет как артефакт в GitLab, потом скачивать его, распаковывать. Это не очень удобно и долго. А если кто-то еще хочет посмотреть отчет у себя, то ему нужно будет проделать те же операции. Нам хотелось получать фидбэк быстрее, и выход нашелся — GitLab pages. Этот встроенная функция, которая доступна из коробки во всех последних версиях GitLab. Позволяет деплоить статические сайты у себя на сервере и получать к ним доступ по прямой ссылке.  
  
Все скриншоты с Allure-отчетами сделаны в GitLab pages. Скрипт для деплоя отчета на GitLab pages — на Windows PowerShell (перед этим необходимо выполнить автотесты):

New-Item -ItemType directory -Path $testresult\history | Out-Null

**try** {Invoke-WebRequest -Uri $hst -OutFile $outputhst}

Catch{echo "fail copy history"}

**try** {Invoke-WebRequest -Uri $hsttrend -OutFile $outputhsttrnd}

Catch{echo "fail copy history trend"}

mvn allure:report

#mvn assembly:single -PzipAllureReport

xcopy $buildlocation\target\site\allure-maven-plugin\\* $buildlocation\**public** /s /i /Y

Что в итоге

Итак, если вы думали о том, нужен ли вам Thread safe-код в Cucumber-фреймворке автотестов, теперь ответ очевиден — с Cucumber 4 его просто внедрить, значительно увеличив тем самым количество запускаемых одновременно потоков. При таком способе запуска тестов вопрос стоит уже о производительности машины с Selenoid и тестового стенда.  
  
Практика показала, что запуск автотестов на тредах позволяет свести расход ресурсов к минимуму при наилучшей производительности. Как видно из графиков, увеличение потоков в 2 раза не приводит к аналогичному ускорению прохождения тестов производительности. Тем не менее мы смогли добавить в сборку приложения более 200 автоматических тестов, которые даже с 5 реранами выполняются примерно за 24 минуты. Это позволяет получать от них быстрый фидбэк, а при необходимости — вносить правки и повторять процедуру снова.