САНКТ - ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Бощенко Алина Вячеславовна

Выпускная квалификационная работа

СОЗДАНИЕ ИНФРАСТРУКТУРЫ ДЛЯ УДАЛЕННОЙ ОТЛАДКИ В ПРИЛОЖЕНИИ НА GRAALVM

Уровень образования: бакалавриат

Направление 01.03.02 «Прикладная математика и информатика»

Основная образовательная программа:

СВ.5004.2017 «Прикладная математика и информатика»

Профиль: «Нелинейная динамика, информатика и управление»

Научный руководитель:

Профессор кафедры прикладной кибернетики

д.ф.-м.н. Юлдашев Марат Владимирович

Рецензент:

Директор ООО «НМТ-Новые мобильные технологии»

Оносовский Валентин Вадимович

Санкт-Петербург

2021 г.

SAINT PETERSBURG STATE UNIVERSITY

Faculty of Mathematics and Mechanics

Boshchenko Alina

Graduation Project

CREATION OF INFRASTRUCTURE FOR REMOTE DEBUGGING IN THE GRAAVVM APPLICATION

Scientific supervisor:

Professor of the Department of Applied Cybernetics,

Dr. of Sci. Yuldashev Marat

Reviewer:

Director of LLC "NMT-New Mobile Technologies"

Onosovsky Valentin

Saint-Petersburg

2021

Благодарность

Выражаем благодарность в адрес Науменко Евгения Владимировича, технического лидера команды “YouTrack”, за кураторство работы от предприятия JetBrains s.r.o.

Оглавление

Введение

Инструменты отладки – это мощный программный инструмент, позволяющий программистам наблюдать за выполнением исследуемой программы, останавливать и перезапускать её, исполнять пошагово, изменять значения в памяти и, в ряде случаев, возвращать назад по времени. В большинстве случаев отладчики используются на этапе разработки программного обеспечения (ПО), однако существует ряд сценариев, при которых необходимо предоставить отладчик не только разработчикам ПО, но и его конечным пользователям. В таких ситуациях возникает необходимость использования удаленной отладки приложения на производственной среде (англ. – *production environments*). Цель удаленной отладки заключается в том, что она позволяет подключиться к приложению, которое исполняется не локально, и отладить это приложение в его среде выполнения. Часто это единственный способ получить те ошибки, которые возникают только на определенном аппаратном обеспечении, что делает создание инструментов для отладки актуальной проблемой. В настоящее время уже существует ряд эффективных отладчиков (Chrome DevTools Protocol, Google, Java Debug Wire Protocol, Oracle см. [9], [10]), однако в данной ситуации они не могут быть применены, так как они не отвечают желаемым требованиям безопасности, не ориентированы на запуск в производственных средах и на использование конечными пользователями. В контексте данной работы используется среда выполнения GraalVM, которая, в отличие от ряда других инструментов, позволяет создать необходимую инфраструктуру для отладки.

GraalVM — это высокопроизводительная среда выполнения, написанная на Java и основанная на OpenJDK. GraalVM поддерживает различные языки программирования и модели выполнения, такие как JIT-компиляция и AOT-компиляция, обеспечивая значительное улучшение производительности и эффективности приложений, что идеально подходит для микросервисов. Среди преимуществ GraalVM важно отметить возможность взаимодействия различных языков программирования в общей среде выполнения. GraalVM исполняет компиляцию байт-кода Java в машинный код, среда может работать как автономно, так и в контексте Node.js или Oracle Database[1]. GraalVM также может работать в контексте OpenJDK для ускорения работы приложений Java с помощью усовершенствованной технологии JIT-компиляции.

1 **Oracle Database** — конвергентная многомодельная система управления базами данных от компании Oracle.

В дополнение к поддержке Java, GraalVM включает новые реализации движков JavaScript, Ruby, R и Python. Они написаны с использованием фреймворка Truffle, который делает возможным создание простых и высокопроизводительных интерпретаторов языков. При написании интерпретатора какого-либо языка с использованием Truffle он автоматически использует Graal для обеспечения JIT компиляции выбранного языка. Таким образом, Graal — не только JIT и AOT компилятор для Java, он также может быть JIT компилятором для JavaScript, Ruby, R и Python.

GraalVM предоставляет возможность удаленной отладки, в том числе кода, написанного на одном языке программирования, внутри кода, написанного на другом языке. Для GraalVM уже существуют готовые отладчики, но их использование существенно затруднено тем, что они ориентированы в первую очередь на разработчиков и не рассчитаны на широкое использование непосредственными пользователями продуктов и системными администраторами, а также сервисными платформами на производственных средах.

Постановка задачи

Было принято решение создать инфраструктуру для удаленной отладки веб-приложения на основе программного обеспечения от компании JetBrains – YouTrack. YouTrack — это веб-платформа для отслеживания задач и управления проектами. YouTrack оптимизирован для разработчиков и Agile-команд и предоставляет широкий ряд инструментов для управления командой и построения процесса разработки. В платформе реализована функциональность, которая позволяет создавать рабочие процессы, написанные на языке JavaScript, для автоматизации задач в процессе разработки программного обеспечения. Рабочие процессы являются основным механизмом расширения и дополнения функциональности YouTrack, доступным пользователям и администраторам системы. В YouTrack версии 2020.5 была реализована поддержка спецификации ECMAScript 2020[2], что значительно упростило взаимодействие пользователя с рабочими процессами. Для обеспечения наиболее полной совместимости со спецификацией ECMAScript 2020 среда выполнения JavaScript (ECMAScript) была перенесена в GraalVM, для которого поддержка возможностей ECMAScript 2020 включена по умолчанию. Учитывая разнообразие и сложность бизнес-процессов разработки программного обеспечения, моделируемых в

2 **ECMAScript 2020** – это одиннадцатое издание спецификации языка ECMAScript. ECMAScript — это встраиваемый, расширяемый, не имеющий средств ввода-вывода язык программирования, используемый в качестве основы для построения других скриптовых языков. Расширения языка включает в себя JavaScript.

YouTrack, рабочие процессы часто достигают большого объема и сложности. В этих условиях их разработка и поддержка в отсутствии отладчика существенно затруднена как на стороне пользователей платформы, так и разработчиков.

Поскольку существующие отладчики ориентированы в первую очередь на разработчиков и на приложения, развернутые не на производственных средах, возникает необходимость разработки промежуточного слоя API в YouTrack для расширения возможностей существующих протоколов отладки, а также сопутствующие инфраструктурные изменения.

Настройка удаленной отладки требует множества шагов по установке и системе безопасности. Эти шаги необходимы, поскольку, если при подключении сеансов отладки к приложениям на серверах возможен доступ к данным других пользователей, расположенных на этом же сервере, нарушаются требования приватности и безопасности, что ведет к серьезным уязвимостям. Ключевыми вопросами являются аутентификация, аудит и, в силу особенности архитектуры платформы, минимизация влияния процесса отладки на YouTrack и его пользователей. Важным аспектом также является обеспечение безопасности персональных данных, хранящихся и обрабатываемых внутри YouTrack согласно GDPR и ряду других актуальных нормативных актов.

Цель работы

Разработка безопасной и высокопроизводительной инфраструктуры для удаленной отладки рабочих процессов в приложении на GraalVM — YouTrack.

Задачи работы

1. Сбор информации и оценка существующих протоколов отладки;
2. Реализация протокола отладки с учетом всех особенностей как облачной, так и on-premise[3] архитектуры YouTrack;
3. Поиск оптимального подхода и реализация аутентификации и ограничения прав доступа при отладке с использованием уже существующих в YouTrack механизмов разграничения прав доступа;
4. Создание изолированного окружения внутри YouTrack для исключения влияния отладчика на стабильную работу сервера;

3 **On-premise** — тип архитектуры, предполагающий использование собственных ресурсов для размещения программного обеспечения.

5. Построение безопасной, но простой в использовании и мониторинге инфраструктуры отладчика, интегрированной в облачную инфраструктуру YouTrack.

Глава 1. Обзор научной литературы и существующих решений

1.1 . Существующие протоколы отладки

В настоящее время на рынке существует ряд протоколов для удаленной отладки приложений, и в данной работе мы рассмотрим два из них – Java Debug Wire Protocol (JDWP) и Chrome DevTools Protocol. Такой выбор обусловлен тем, что GraalVM поддерживает отладку приложений и по умолчанию предоставляет встроенную реализацию протокола Chrome DevTools. При ряде дополнительных настроек это позволяет подключать к GraalVM совместимые отладчики, такие как Chrome Developer Tools. JDWP применяется для отладки кода непосредственно из среды разработки (IDE), что делает процесс значительно удобнее. Подключение к коду осуществляется из IDE с использованием сетевого подключения TCP и для выполнения отладки IDE необходимо иметь исходный код для запущенных классов.

Java Debug Wire Protocol – это протокол, который соединяет Java Virtual Machine Tool Interface с консолью отладчика, вне зависимости от того, является ли эта консоль локальной или удаленной. Java Virtual Machine Tool Interface - это программный интерфейс для взаимодействия с JVM. Он предоставляет способ как проверки состояния, так и управления исполнением приложений, работающих на JVM. Также он поддерживает весь спектр инструментов, которым необходим доступ к состоянию JVM, включая профилирование, отладку, мониторинг и анализ потоков. Диаграмма развертывания (англ. – *deployment diagram*) для приложений с JDWP обычно выглядит следующим образом:

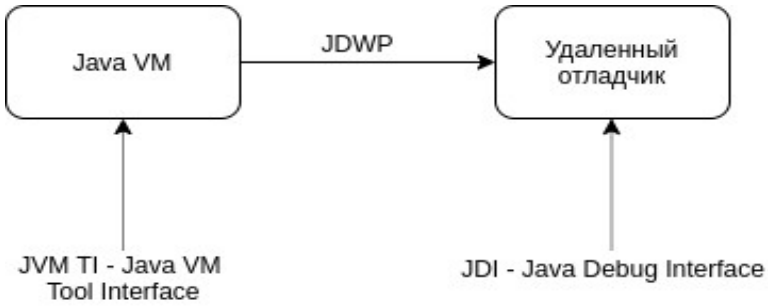

Рисунок 1: Диаграмма развертывания с JDWP

Принцип работы: JDWP является проводным протоколом (англ. – *wire protocol*). Проводной протокол - это способ передачи данных по каналу между различными

вычислительными средами. В отличие от сетевых протоколов на транспортном уровне (например, TCP или UDP), термин «проводной протокол» используется для описания способа представления информации на уровне приложения. Для такого представления требуется общий информационный набор (например, XML) и привязка данных (например, с использованием общей схемы кодирования, такой как XSD). Проводные протоколы соединяют несколько платформ, некоторые из них не зависят от языка, что позволяет передавать программы, написанные на разных языках.

Протокол основан на официальной спецификации от компании Oracle. Для установки первоначального рукопожатия, которое инициирует соединение, отладчик отправляет 14 байтов, содержащих 14 ASCII символов строки «JDWP Handshake», виртуальная машина отправляет в ответ те же самые 14 байт «JDWP Handshake». После этого связь между отладчиком и самой виртуальной машиной Java происходит посредством обмена пакетов команд, отправленных отладчиком, и ответных пакетов, отправленных программой. Пакеты команд используются отладчиком для запроса информации от виртуальной машины, например, о точке остановки или исключении, или для управления выполнением программы, включая вызов методов внутри виртуальной машины. Ответный пакет отправляется только в ответ на пакет команды и всегда предоставляет информацию об успешном или неудачном выполнении команды. Ответные пакеты также могут нести данные, запрошенные в команде (например, значение поля или переменной). В то же время события, отправленные с виртуальной машины, не требуют ответного пакета от отладчика.
Протокол является асинхронным -- несколько пакетов команд могут быть отправлены до того, как будет получен первый ответный пакет.

Таким образом, благодаря способности создавать пакеты, которые позволяют взаимодействовать с виртуальной машиной Java, становится возможным заставить виртуальную машину Java загружать произвольные классы Java в память и использовать их для удаленного выполнения кода, что может оказаться серьезной уязвимостью.

У протокола JDWP нет шифрования и аутентификации пользователей, что делает его небезопасным, причем для использования как на облачной, так и на on-premise архитектуре. Однако в производственной среде в большинстве случаев функция отладки по умолчанию отключена, а значит, отсутствие шифрования и аутентификации протокола не является угрозой ослабления безопасности. Однако для целей данной работы мы хотим добиться прямо противоположного – использования отладчика в производственной среде.

Возможности протокола:

- Исполнение статических функций
- Исполнение функций из экземпляра объекта в JVM
- Возможность JVM работать в определенном режиме, задаваемом командой. Позволяет ставить точки остановки и исполнять код пошагово.
- Появление оповещений в стандартных отладчиках (GDB, WinDBG) по событиям доступа к переменным или их изменения
- Доступ к логам, выводимым в консоль приложения, и их просмотр

Рассмотрим **Chrome DevTools Protocol** и его отличия от предыдущего протокола. Протокол Chrome DevTools предоставляет инструменты для проверки и отладки Chromium, Chrome и других браузеров на основе Blink – браузерного ядра (англ. – *browser engine*), разработанного компанией Google.

Принцип работы: Как и JDWP, Chrome DevTools Protocol является проводным протоколом. Сторонние клиенты, такие как IDE, редакторы, средства непрерывной интеграции и фреймворки тестирования, могут интегрироваться с отладчиком Chrome для отладки и предварительного просмотра кода, а также управления браузером. В настоящее время протокол отладки Chrome поддерживает только одного клиента на страницу. Таким образом, можно использовать либо, например, инструмент DevTools[4] для проверки страницы, либо некоторый сторонний клиент, но не одновременно.

Приложения, работающие в Chrome (например, веб-IDE[5]), могут создавать расширение Chrome с помощью модуля отладчика chrome.debugger. Этот модуль позволяет расширению напрямую взаимодействовать с отладчиком, минуя пользовательский интерфейс DevTools. Однако нас интересует отладка в локальной IDE, и для этого приложение может использовать проводной протокол Chrome DevTools для прямой интеграции с отладчиком. Этот протокол включает обмен сообщениями JSON через соединение WebSocket[6].

4 **Chrome DevTools** – это набор инструментов для отладки и диагностики, встроенных непосредственно в браузер Google Chrome, отладка осуществляется с помощью Chrome DevTools Protocol.

5 **IDE** — интегрированная среда разработки

6 **WebSocket** — протокол связи поверх TCP-соединения, предназначенный для обмена сообщениями между веб-сервером и браузером в режиме реального времени

В отличие от протокола JDWP, Chrome DevTools протокол предоставляет инструменты для осуществления аутентификации. Для этого достаточно выставить соответствующий заголовок в запросе.

Возможности протокола:

- Исполнение и отладка JavaScript
- Инспекция элементов в дереве DOM
- Редактирование элементов и CSS на лету
- Проверка и мониторинг производительности сайта/приложения
- Имитация местонахождения пользователя
- Имитация скорости сети
- Доступ к логам, выводимым в консоль приложения, и их просмотр

Глава 2. Подключение протокола отладки к приложению на GraalVM

2.1 . Архитектура части приложения, подлежащей отладке

Рабочий процесс определяется как набор файлов, написанных на языке JavaScript. Некоторые файлы экспортируют объекты со специальными свойствами, которые сообщают среде выполнения о том, что объект определяет правило. К примеру, функция Issue.onChange определяет следующие свойства: тип правила – "onChange" и цель – "Issue".

```
exports.rule = Issue.onChange({
        title: 'Some title',
        guard: function (ctx) {
                return true;
        },
         action: function (ctx) {
                // do something
        }
});
```

При сохранении сценария слушатели запускают обработку сценария. Это включает в себя транзитивный вызов метода, который на основе свойств экспортируемых свойств объекта понимает, что объект является правилом рабочего процесса, которое должно иметь несколько обязательных свойств (например, действие). Эта часть среды выполнения создает экземпляр класса для представления правила. Такой экземпляр будет называться экземпляром правила.

После создания экземпляра правила он сохраняется в памяти в двух различных bean-компонентах. Разница между ними заключается в том, что первая используется для содержания объектов правил, созданных из предопределенных рабочих процессов (то есть тех, которые включены в дистрибутив YouTrack), тогда как последняя содержит объекты правил, созданные из настраиваемых рабочих процессов и предопределенных объектов правил.

Общей частью вызова правила любого типа является ввод контекста GraalVM. Каждое правило инкапсулирует пул контекстов, из которых можно получить контекст. Для правил, которые выполняются при изменениях определенных частей приложения, метод вызывается, когда управляющий класс правил реагирует на изменения, внесенные в задачу (или такие объекты, как комментарий, вложение, поле задачи) в транзакции. Поскольку рабочий процесс может дополнительно изменять состояние задачи, управляющий класс вызывает рабочие процессы в цикле, предлагая им отреагировать на новые изменения.

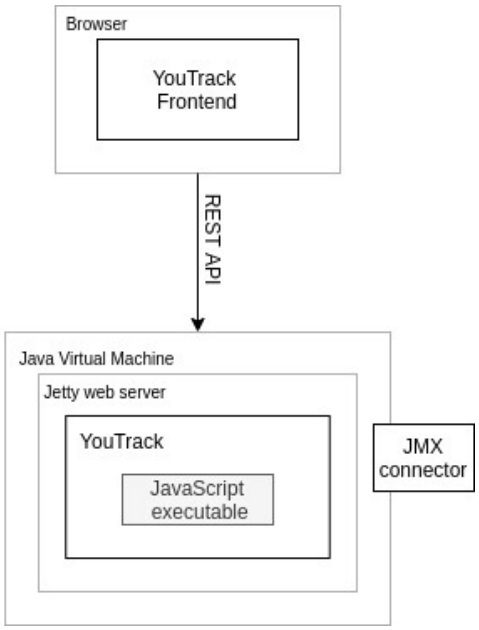

Рисунок 2: Архитектура части приложения, подлежащей отладке

Кодовая база проекта написана на языках Java и Kotlin, сборка осуществляется с помощью системы автоматической сборки Gradle. Gradle основан на графе задач (task), которые могут зависеть друг от друга. Задачи предоставляются различными плагинами (plugins), а стандартным языком их описания является Groovy DSL. Одной из главных задач системы сборки является управление зависимостями, то есть определением того, какие библиотеки и фрэймворки нужны проекту.

2.2 . Подключение протокола Chrome Debug Protocol

Для установки первичного подключения удаленного протокола отладки был выбран Chrome Debug Protocol, в силу того, что GraalVM поддерживает отладку приложений и предоставляет встроенную реализацию этого протокола Truffle Chrome Inspector. Это

позволяет нам подключать к GraalVM совместимые отладчики, такие как, например, Chrome Developer Tools.

Подключение необходимых зависимостей

Для корректной работы Chrome Debug Protocol в приложении необходимо добавить соответствующую зависимость в файл конфигурации build.gradle, а именно:

```
dependencies {
    compile lib: 'org.graalvm.tools', name: 'chromeinspector', version: '20.2.0'
    ...
}'[7].
```

Это позволит подключить необходимую библиотеку на этапе компиляции проекта.

Настройка корректной работы протокола

Поскольку скрипты, подлежащие отладке, написаны на языке JavaScript, в приложении используется специальный инструмент для их исполнения — Engine. Engine — это механизм выполнения гостевых языков в Graal, позволяющий проверять установленные гостевые языки, инструменты и их доступные опции.

Для подключения протокола на этапе создания объекта Engine необходимо было определить его параметры, выбранные на основе официальной документации GraalVM от компании Oracle (Приложение 7. ScriptingContextFactory.kt):

1. **inspect=host:port.** Данная опция запускает Chrome Inspector на заданном хосте и порте.
2. **inspect.Suspend=false.** Данная опция необходима для предотвращения остановки программы при запуске первой исходной строки, по умолчанию присваивается true.
3. **inspect.Secure=false.** Данная опция отвечает за использование TLS / SSL, по умолчанию присваивается false для адреса обратной связи, в противном случае — true. На данном этапе работы эта опция необходима, так как без нее необходимо дополнительно проводить процедуру аутентификации, что и было сделано в дальнейшем (см. Глава 3). На данном этапе подключение не является безопасным, а также позволяет подключиться к отладке рабочих скриптов любому пользователю.

7 Maven repository: https://mvnrepository.com/artifact/org.graalvm.tools/chromeinspector

Задав такие настройки, при запуске YouTrack одновременно с ним появляется возможность открыть Google DevTools по уникальному адресу в формате devtools://devtools/bundled/js_app.html?ws=[host]:[port]/[key] для удаленной отладки приложения. Таким образом, первоначальное подключение протокола отладки произведено.

Для организации такой инфраструктуры в реальной производственной среде необходимо решить ряд следующих вопросов:

1. Введение обязательной аутентификации пользователя;
2. Фильтрация доступных для отладки пользователем скриптов, согласно его правам доступа;
3. Настройка протокола таким образом, что отладка происходит на отдельном потоке, не влияя на остальные процессы приложения;
4. Повышение устойчивости облачных экземпляров YouTrack путем ограничения времени жизни точек останова.

Глава 3. Настройка аутентификации протокола отладки

Выбор наилучшего решения для организации структуры аутентификации является одним из наиболее объемных разделов исследовательской части данной работы, и в последующих трех разделах представлены экспериментальные решения, которые ведут к выбору наиболее оптимального подхода.

3.1. Настройка SSL канала аутентификации

В Truffle Chrome Inspector, который является инструментом отладки, реализованным в GraalVM и используется для данной работы, за безопасность подключения отвечает ряд следующих опций, определяющихся на этапе создания объекта класса Engine:

1. **inspect.Secure**=false|true

   При значении опции, равном true, TLS/SSL используется для защиты передачи данных по протоколу. Помимо изменения ws (web socket) на wss (web socket secure), конечная точка HTTP (HTTP endpoint), которая предоставляет метаданные об отлаживаемом приложении, также изменяется на HTTPS. Это несовместимо, например, со страницей chrome://inspect, которая не может предоставить информацию об отладке и запустить отладчик. В этом случае отладка должна запускаться напрямую через URL-адрес WSS. Также необходимо использовать стандартные системные параметры javax.net.ssl.* чтобы предоставить информацию о хранилище ключей с ключами шифрования TLS / SSL, или определить значения следующих опций:
2. **inspect.KeyStore –** путь к файлу хранилища ключей
3. **inspect.KeyStoreType –** тип файла хранилища ключей (по умолчанию JKS)
4. **inspect.KeyStorePassword –** пароль хранилища ключей
5. **inspect.KeyPassword –** пароль для восстановления ключей, если он отличается от пароля хранилища ключей

В данном подходе для защиты соединения были выставлены значения вышеописанных параметров. Таким образом, защищенное SSL-соединение было установлено, однако впоследствии было принято решение отказаться от осуществления аутентификации пользователя этим методом по следующим причинам:

- Настройка параметров SSL-KeyStore требует определенных навыков, которыми владеют не все пользователи продукта, а значит для них это может стать препятствием в использовании отладчика
- Ряд пользователей используют самоподписанные SSL-сертификаты, что добавляет сложности в процесс настройки безопасности
- С точки зрения построения рабочего процесса, более понятного и удобного для пользователя, логичнее реализовать систему аутентификации в связке с аутентификацией самого приложения YouTrack

3.2. Интеграция отладчика в YouTrack в качестве модуля

Помимо настройки SSL, во встроенной в Graal реализации протокола Truffle Chrome Inspector не было предусмотрено каких-либо средств аутентификации пользователя. Поэтому для решения задачи аутентификации возникла необходимость модифицировать часть отладчика для включения системы аутентификации на этапе установки соединения.

Для этого было принято решение интегрировать отладчик в систему не как стороннюю библиотеку, подключаемую через зависимость, а как проектный модуль.

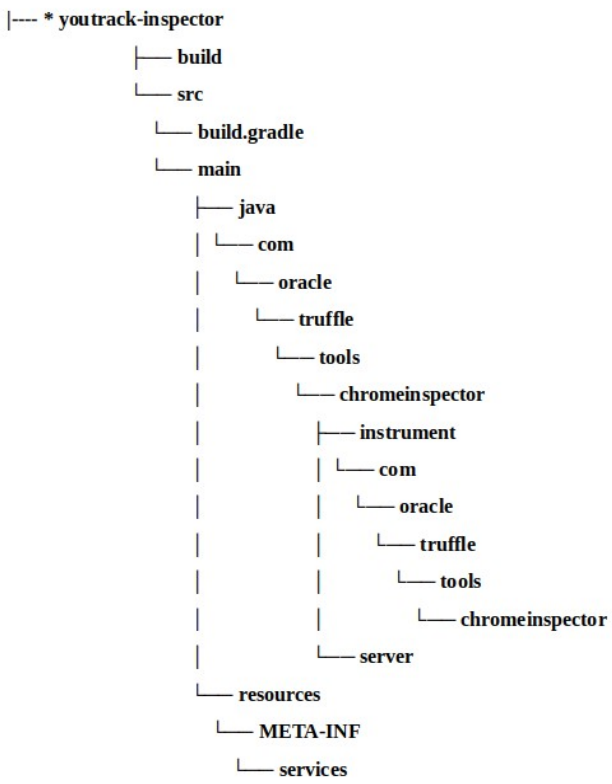

Рисунок 3: Структура модуля отладчика в проекте YouTrack

В ходе работы стало понятно, что для достижения цели достаточно модифицировать несколько файлов из библиотеки, поэтому для упрощения дальнейшей поддержки и уменьшения объема модуля было принято решение подключить отладчик как стороннюю библиотеку и создать собственный инструмент отладки путем включения в модуль проекта ряда модифицированных компонент исходного инструмента.

После создания собственного модуля отладчика в приложении YouTrack, (Рисунок 3) определяемые параметры объекта класса Engine приняли следующий вид:

- **workflowsInspector=host:port**
- **workflowsInspector.Secure=false**
- **workflowsInspector.Suspend=false**

Данные параметры были определены при инициализации модифицированного инструмента (Приложение 1. YouTrackInspectorInstrument.java) и сгенерированы в качестве опций сервиса DSL-процессором (Приложение 6. YouTrackInspectorInstrumentOptionDescriptors.java). DSL процессор (Truffle DSL Processor) – это аннотационный процессор, который генерирует исходный код из узлов, которые объявлены с использованием DSL. DSL – это предметно-ориентированный язык, позволяющий декларативно указывать специализированные узлы абстрактного синтаксического дерева[8]. Он использует процессор аннотаций для создания необходимого подкласса. В данном случае, для того, чтобы процессор мог сгенерировать опцию отладчика, необходимо явно указать ее с использованием аннотации @com.oracle.truffle.api.Option.

В ходе реализации решения поставленной задачи возник ряд сложностей, на которые следует обратить внимание:

1. Созданный модуль с модифицированным инспектором был создан, но не был обнаружен системой, что привело к отсутствию подключенного отладчика и выдаче диагностических сообщений о не найденных опциях. Данная проблема была решена путем отслеживания трассировки стека из отладчика, и было выяснено, что

8 **Дерево абстрактного синтаксиса** — конечное помеченное ориентированное дерево, внутренние вершины которого сопоставлены с операторами языка программирования, а листья — с соответствующими операндами. **Service Provider Interface** (**SPI, Интерфейс поставщика услуг**) - это API, предназначенный для реализации или расширения третьей стороной, использующийся для включения расширения каркаса и сменных компонентов.

она возникает из-за отсутствия реализации SPI[9]. Для успешного определения инструмента у провайдера, в нашем случае – TruffleInstrument.Provider, необходимы дескрипторы, сгенерированные DSL-процессором в файле YouTrackInspectorInstrumentOptionDescriptors.java и SPI инструмента (Приложение 2. YouTrackInstrumentProvider.java) с соответствующим файлом META-INF/services/com.oracle.truffle.api.instrumentation.TruffleInstrument$Provider, определяющим его в качестве сервиса TruffleInstrument.Provider.

2. Созданная директория META-INF/services не была обнаружена в jar модуля. Проблема была решена модификацией файла конфигурации build.gradle и перемещением META-INF/services в директорию youtrack-inspector/src/resources
3. После написания SPI инструмент был обнаружен провайдером, однако, его опции по-прежнему не были найдены. Используя отладку, в частности точки останова и трассировку стека (stacktrace), стало понятно, что обнаруженные хэши опций были неверны. После изменения хэшей опций на корректные проблема была решена.

```
plugins {
  id "com.github.lokkie.gradle.serviceloader" version "1.1.7"
}
jar {
  from ('resources') {
    include 'META-INF/services/*.*'
  }
  from ('./src/main/resources') {
    include 'META-INF/services/com.oracle.truffle.api.instrumentation.TruffleInstrument$Provider'
  }
}
```

Рисунок 4: Модификация конфигурации build.gradle

3.3. Настройка аутентификации в синхронизации с аутентификацией приложения YouTrack

Принцип работы с подключением к протоколу отладки

В исходной реализации отладчик открывает как WebSocket-, так и HTTP-подключение, разворачивает свой собственный сервер. После запуска отладчика на стороне сервера, при запросе с клиента на конечную точку **http://debughost:debugport/json** возвращается ответ в формате JSON с информацией об адресе вебсокета, на котором запущен отладчик. В качестве ответа возвращаются данные с открытыми в браузере Chrome вкладками и

9 **Service Provider Interface** (**SPI, Интерфейс поставщика услуг**) - это API, предназначенный для реализации или расширения третьей стороной, использующийся для включения расширения каркаса и сменных компонентов

адресом вебсокета (**webSocketDebuggerUrl**), по которому можно осуществить подключение.

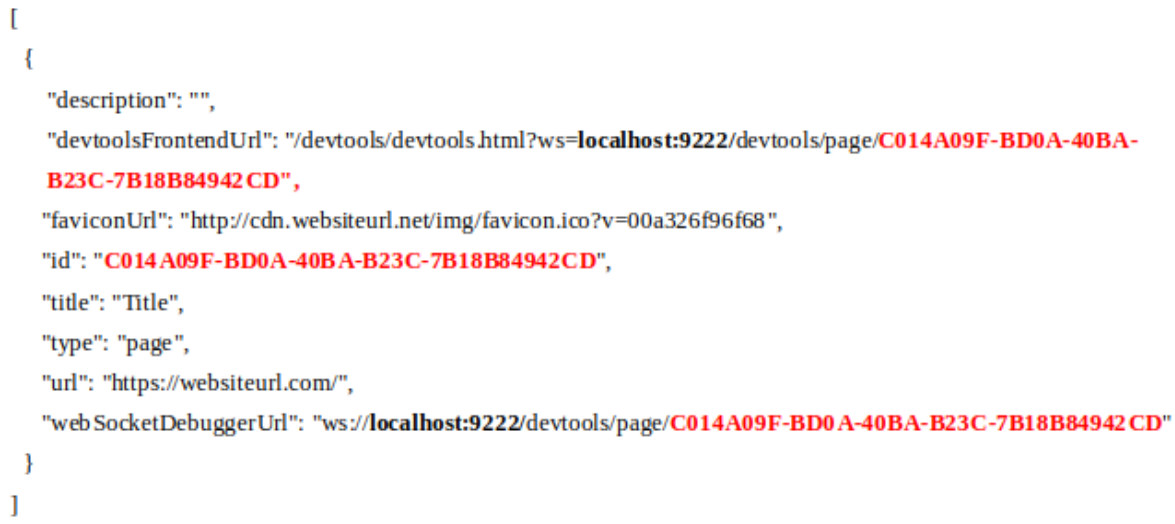

Рисунок 5: Пример ответа с конечной точки http://host:port/json

Рассмотрим процесс подключения к отладчику со стороны клиента. При установке подключения клиент выполняет запрос на **http://debughost:debugport/json,** где debughost и debugport — хост и порт отладчика. Из полученного ответа извлекается **webSocketDebuggerUrl,** по которому непосредственно и осуществляется подключение.

С точки зрения производственной среды создание отдельного порта для отладки в данном случае может вызвать ряд затруднений, так как сетевые настройки машины пользователя неизвестны (в частности, наличие или отсутствие обратного прокси-сервера (reverse proxy server)[10], межсетевого экрана (firewall)[11] и балансировки нагрузки (load balancing)[12]). Также при рассмотрении облачной архитектуры мы сталкиваемся с тем, что там могут быть задействованы несколько экземпляров YouTrack в пределах одного процесса. В этом случае любая новая реализация HTTP должна будет обладать способом отделения одного экземпляра от другого.

10 **Обратный прокси-сервер** — тип прокси-сервера, который ретранслирует запросы клиентов из внешней сети на один или несколько серверов, логически расположенных во внутренней сети.

11 **Межсетевой экран** — программный или программно-аппаратный элемент компьютерной сети, осуществляющий контроль и фильтрацию проходящего через него сетевого трафика в соответствии с заданными правилами.

12 **Балансировка нагрузки** — метод распределения заданий между несколькими сетевыми устройствами (пример — сервера) с целью оптимизации использования ресурсов, сокращения времени обслуживания запросов, горизонтального масштабирования кластера, а также обеспечения отказоустойчивости.

Более того, в архитектуре приложения YouTrack реализован специальный сервлетный фильтр, который, преобразовывая содержание HTTP-запросов и информацию, содержащуюся в заголовках запроса, позволяет выполнять проверку аутентификации при каждом запросе. Фильтры сервлетов — это классы Java, которые можно использовать для того, чтобы перехватывать запросы от клиента, прежде чем они получат доступ к ресурсу на стороне сервера, и манипулировать ответами от сервера, прежде чем они будут отправлены обратно клиенту. Подключаемые фильтры являются оптимальным решением для стандартной обработки сервисов, не требуя при этом изменений в основном коде обработки запросов. Фильтры перехватывают входящие запросы и исходящие ответы, обеспечивая пред- и постобработку. Фактически, в основное приложение добавляется множество общих служб, таких как безопасность, ведение журнала и так далее. Эти фильтры представляют собой компоненты, не зависящие от основного кода приложения, и они могут быть добавлены или удалены декларативно.

Таким образом, ни один запрос к любой конечной точке приложения не может пройти, миновав этап аутентификации пользователя, что может быть использовано и для аутентификации пользователя при подключении отладчика. Так как именно на данном этапе проверяется соответствие логина и пароля/токена пользователя и его права в системе, а работает данный фильтр только для конечных точек приложения YouTrack, нам критически важно получить JSON с информацией о вебсокете именно с конечной точки приложения.

Основная идея подхода заключается в том, что при подключении клиента debughost и debugport используются только для запроса на конечную точку **/json,** а значит, если настроить конечную точку **http://youtrackhost:youtrackport/json** так, что она в качестве ответа будет выдавать всю необходимую информацию об отладчике, то и в конфигурации можно будет указать не **debughost и debugport, а youtrackhost и youtrackport.**

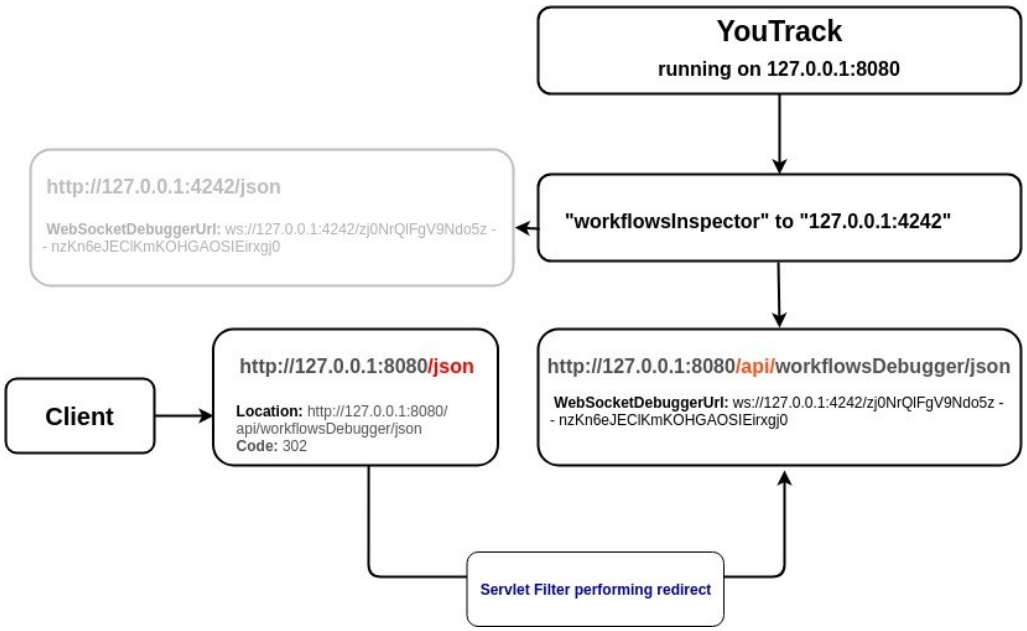

Рисунок 6: Архитектура связи клиента, отладчика и приложения YouTrack с использованием сервлетного фильтра, где в качестве примера youtrackhost:youtrackport = 127.0.0.1:8080, a debughost:debugport = 127.0.0.1:4242

Однако в силу архитектуры API приложения, создавать конечные точки можно только относительно корневого URL **http://youtrackhost:youtrackport/api/,** что осложняет ситуацию, поскольку конфигурация отправляет запрос по другому адресу.

Для решения данного вопроса мы применили технологию сервлетных фильтров. Был создан специальный сервлетный фильтр (Приложение 4. DebugServletFilter.java), который позволяет перенаправлять запрос с **http://youtrackhost:youtrackport/json** на **http://youtrackhost:youtrackport/api/workflowsDebugger/json,** который содержит всю необходимую информацию о вебсокете.

Однако впоследствии выяснилось, что не во всех клиентах инструмента отладки поддерживается перенаправление, вследствие чего фильтр использоваться не может. Замена на переадресацию также не принесла результата, так как из-за многопоточности приложения терялся корректный контекст. Так как изначально предполагалось, что в качестве основного клиента будет использоваться среда разработки (YouTrack IDE Plugin), было принято решение написать специальную конфигурацию для подключения к конечной точке с информацией о вебсокете, основываясь на следующих соображениях:

- Возможность подключиться непосредственно к конечной точке **youtrackhost:youtrackport/api/workflowsDebugger/json** без использования фильтра, так как мы самостоятельно пишем реализацию подключения;
- Поскольку в текущей конфигурации не предусматривается возможность авторизации и нет средств для ее реализации, даже при правильном подключении пользователь не сможет осуществить авторизацию;
- Созданную конфигурацию можно будет включить в YouTrack IDE Plugin[13], который интегрирует ее непосредственно в среду, что позволит значительно улучшить пользовательский опыт.

Данная реализация подробно описана в разделе 3.6. Однако, несмотря на то, что мы можем обойти вопрос аутентификации, реализовав ее путем проверки через сервлетный фильтр, это решение не покрывает задачу выдачи файлов для отладки на основе разрешений пользователя в системе.

Таким образом, мы приходим к наиболее эффективному решению поставленных задач – реализации собственного WS-соединения. С его помощью становится возможной выдача файлов для отладки на основе разрешений пользователя в системе. Более того, при интеграции вебсокетного соединения в сервер приложения YouTrack, управлять SSL-аутентификацией отладчика будет тоже YouTrack, а значит, все пользовательские настройки могут быть использованы. И наконец, только после реализации собственного WS-соединения будет возможно изменить систему работы точек останова и повлиять на нее (раздел 4.1).

3.4. Реализация собственного WS-соединения

Для реализации функциональности выдачи файлов на основе разрешений пользователя в системе и ограничения времени жизни точек останова для минимизации возможности перегрузки сервера была разработана модифицированная версия вебсокетного соединения. В исходной реализации отладчик создает свой собственный сервер на [debughost:debugport], на котором осуществляется передача данных как по HTTP, так и по WS. Однако, создание отдельного сервера может не сработать в условиях пользовательской системы в силу ряда ограничений, о которых упоминалось ранее, таких, как обратный прокси-сервер, межсетевой экран и балансировка нагрузки. Само приложение YouTrack интегрируется в систему пользователя с учетом всех факторов и настраивается отдельно, но данные настройки YouTrack не могут быть перенесены на

13 YouTrack IDE Plugin — плагин, позволяющий работать с приложением YouTrack из IDE

отладчик при текущей реализации в силу того, что он запускается на отдельном сервере на собственном хосте и порте. Более того, HTTP соединение необходимо только для того, чтобы получить доступ к конечной точке **/json,** по которой будет получен адрес вебсокета, по которому отладчик будет взаимодействовать с системой. Но поскольку данная конечная точка была перенесена в YouTrack (Раздел 3.3), необходимость открытия HTTP отпадает. Таким образом, перед нами встает задача интеграции WS-соединения в сервер приложения YouTrack с последующим удалением дополнительного сервера отладчика.

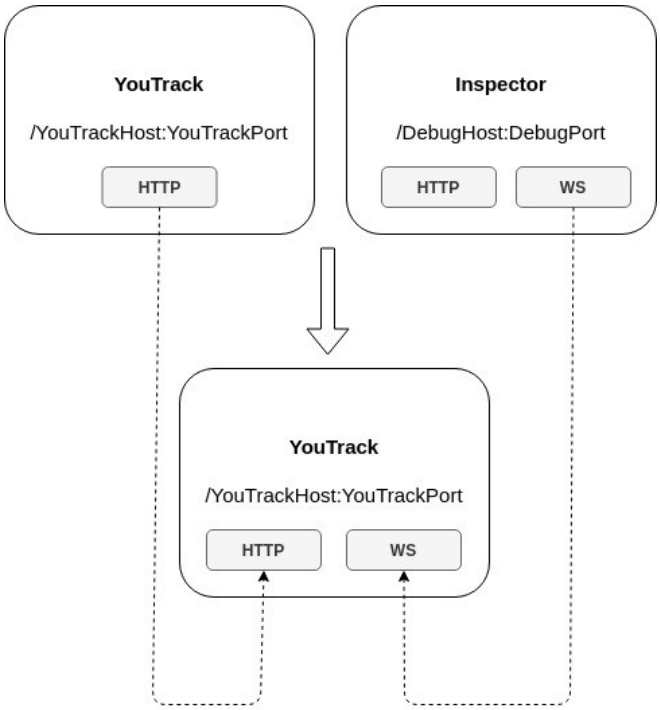

Рисунок 7: Переход от архитектуры с отдельным сервером для отладчика к архитектуре с интегрированным в YouTrack WS-соединением

Для начала необходимо перенести существующую реализацию соединения в проект YouTrack без потери функционала и удалить дополнительный сервер отладчика. Это достигается при помощи создания сервлета (Приложение 3. WorkflowsDebuggerServlet.kt), представляющего собой интерфейс, реализация которого расширяет функциональные возможности сервера приложения, а именно отвечает за интеграцию вебсокета. Логика обработки сообщений реализована в классе DebuggerMessagingAdapter (Приложение 8. DebuggerMessagingAdapter.kt), и таким образом мы интегрируем вебсокет, по которому отладчик будет взаимодействовать с системой, в сервлетную архитектуру приложения. Финальная реализация инструмента представлена в Приложении 10. DebuggerInstrument.kt

3.5. Выдача файлов для отладки согласно правам пользователя в системе

Благодаря интеграции вебсокетного взаимодействия отладчика с файлами в собственный сервер приложения, появилась возможность организации инфраструктуры для использования клиентских инструментов отладки (Chrome Developer Tools / YouTrack IDE Plugin) и реализации системы аутентификации пользователя и выдачи ему скриптов для отладки согласно правам в системе. Для начала процесса отладки скриптов автоматизации необходимо подключить клиента отладчика. Рассмотрим в качестве примера Chrome Developer Tools.

При подключении клиента отладки необходимо выполнить запрос на конечную точку [youtrackHost]:[youtrackPort]/api/workflowsinspector/json. На данном этапе происходит первый шаг аутентификации – проверяются права пользователя на чтение и создание новых скриптов автоматизации. Для просмотра, создания и редактирования скриптов рабочих процессов требуется, чтобы у пользователя были разрешения на чтение проекта и обновление проекта по крайней мере в одном проекте. Пользователи с полномочиями низкоуровневого администрирования могут создавать рабочие процессы и редактировать любые рабочие процессы в системе.

Данная проверка выполняется с помощью сервлетного фильтра авторизации в приложении, поскольку путь */api/* входит в его поле действия. Таким образом на этом этапе мы получаем пользователя с его правами в системе.

Ответ на данный запрос возвращается в формате JSON:

- Для авторизованного пользователя с необходимыми правами:


```
[{
"devtoolsFrontendUrl":
"devtools://devtools/bundled/js_app.html?ws=[host]:[port]/debug/o0nvsnvLe3uOEVr8XTVU5zj7QrYvEug5HysJ10Fqhck",
"description": "GraalVM",
"webSocketDebuggerUrl":
"ws://[host]:[port]/debug/o0nvsnvLe3uOEVr8XTVU5zj7QrYvEug5HysJ10Fqhck"
}]
```


- Для неавторизованного пользователя или пользователя без необходимых прав:


```
{
```

"error": "Unauthorized",
"error_description": ""
}

Ключевыми полями здесь являются поля webSocketDebuggerUrl и devtoolsFrontendUrl. Они используются для последующего подключения YouTrack IDE Plugin и ChromeDevTools соответственно. Перед формированием этого ответа для пользователя формируется собственный токен безопасности, который должен быть использован при инициализации сессии отладки и только один раз — в данном случае это **«o0nvsnvLe3uOEVr8XTVU5zj7QrYvEug5HysJ10Fqhck».** Основными причинами принятия данного решения послужили необходимость предотвращения атаки перебором[14], и возможность отлаживать скрипты на одном приложении нескольким пользователям. При использовании одного и того же токена для всех пользователей и всех сессий отладки при наличии ресурсов становится достаточно легко подобрать токен, а следовательно, получить доступ ко всей системе. На этом этапе формируется хэш-таблица, ключами в которой являются пользователи с их правами, а значениями – сгенерированные для них токены.

Затем при непосредственном открытии ChromeDevTools / YouTrack IDE Plugin происходит второй шаг аутентификации пользователя. ChromeDevTools могут быть открыты в качестве клиента отладчика непосредственно по адресу devtoolsFrontendUrl, указанному в JSON-ответе. Токен из адреса проверяется на наличие в хэш-таблице и соответствие некоторому пользователю. Таким образом на данном этапе мы получаем соответствующего пользователя с его правами в системе и можем выдать доступные файлы. Так как токен может быть использован только один раз даже для того же самого пользователя, после подключения клиента отладчика он удаляется из таблицы. Для того, чтобы переподключиться к отладчику, клиенту необходимо заново отправить запрос на конечную точку /api/workflowsinspector/json и получить адрес, содержащий новый токен.

Для клиента на базе конфигурации YouTrack IDE Plugin алгоритм подключения будет аналогичным, за исключением того, что используется не devtoolsFrontendUrl а webSocketDebuggerUrl.

14 **Перебор по словарю** — атака на систему защиты, использующая метод полного перебора (англ. brute-force) предполагаемых паролей, используемых для аутентификации, осуществляемого путём последовательного пересмотра всех слов определённого вида и длины из словаря с целью последующего взлома системы. Любая задача из класса NP может быть решена полным перебором.

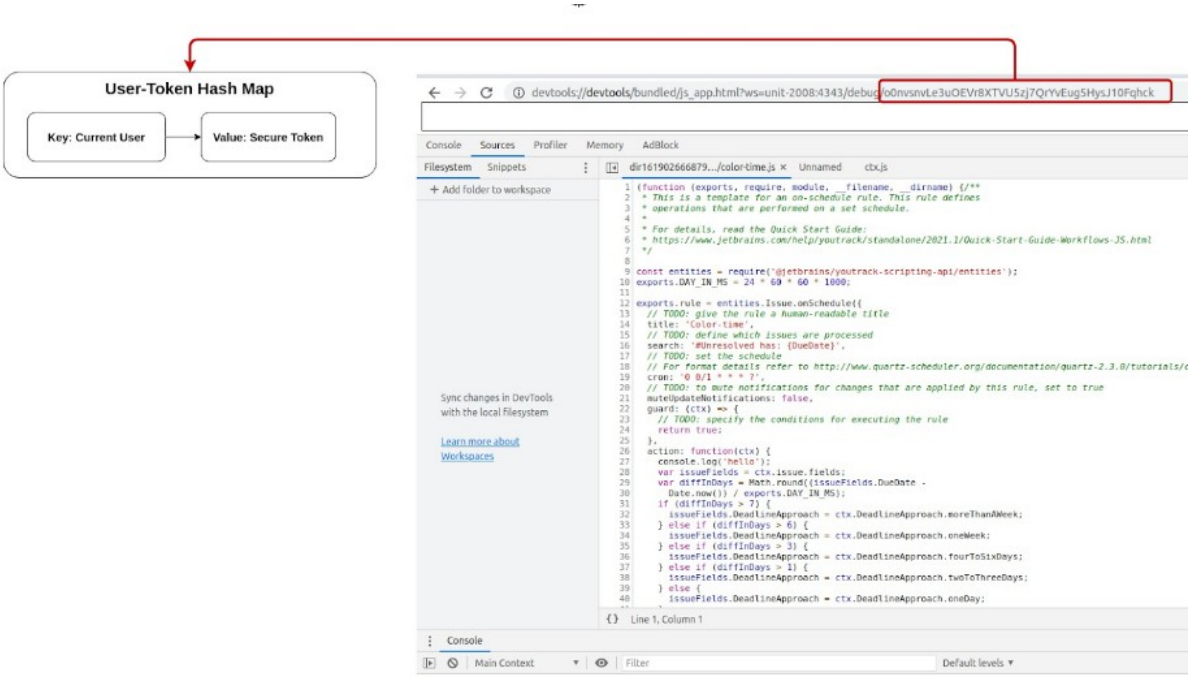

Рисунок 8: Организация инфраструктуры для использования клиентских инструментов отладки

3.6. Подключение IDE в качестве клиента инструмента отладки

Использование конфигурации удаленной отладки в IDE может в значительной степени улучшить пользовательский опыт, так как с помощью нее можно синхронизировать удаленные и локальные скрипты и отлаживать их непосредственно в IDE, не используя дополнительных клиентов, таких как Chrome Developer Tools.

В IDE, базирующихся на платформе IntelijIDEA Ultimate, уже существует конфигурация для удаленной отладки с помощью Chrome Debug Protocol. Это конфигурация Attach to Chrome/NodeJS, на основе которой и была написана конфигурация для отладки рабочих скриптов.

Для того, чтобы подключиться к инструменту отладки, необходимо определить хост и порт в конфигурации и явно прописать все связи соответствующих локальных и удаленных файлов, для того, чтобы редактировать реальные, а не скомпилированные файлы, открытые только на чтение. Также необходимо ввести токен аутентификации для получения доступа к функционалу отладки. Токен генерируется при создании учетной записи пользователя приложения YouTrack и является его уникальным идентификатором. Особое внимание необходимо уделить расширению JetBrains for Chrome при его наличии – порт в расширении должен совпадать с портом, определенным в конфигурации и, соответственно, в самом отладчике.

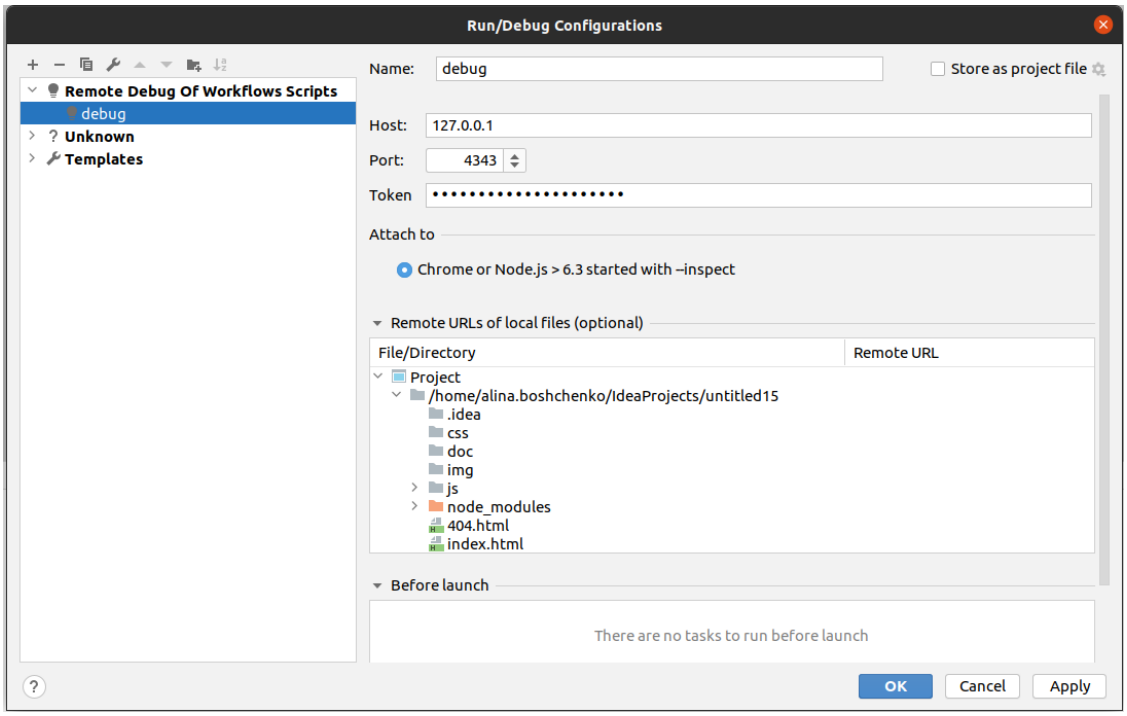

Рисунок 9: Созданная конфигурация Remote Debug Of Workflows Scripts

Осуществление процедуры аутентификации

При запуске конфигурации выполняется запрос на конечную точку **youtrackhost:youtrackport/api/workflowsDebugger/json,** и по полученному адресу вебсокета webSocketDebuggerUrl осуществляется подключение к инструменту отладки в качестве клиента. Подробная реализация подключения показана в Приложении 7. JSRemoteWorkflowsDebugConfigurationType.kt. Конфигурация интегрируется в IDE IntelijIDEA путем создания класса ее типа и регистрация его в plugin.xml. Реализация взаимодействия IDE как клиента с инструментом отладки представлена в Приложение 5. WipConnection.kt.

Таким образом мы получаем устойчивую и безопасную реализацию аутентификации пользователя YouTrack в инфраструктуре отладки, а также выдаем скрипты автоматизации для отладки согласно правам пользователя в системе.

Глава 4. Повышение устойчивости приложения к новой функциональности

4.1. Ограничение времени жизни точек останова для минимизации возможности перегрузки сервера

Учитывая особенности облачной архитектуры приложения, стабильная работа серверов является критически важным моментом разработки. Точки останова являются одной из ключевых функций отладчика. Необходимо не допустить ситуацию, в которой наличие паузы исполнения скрипта на активной точке останова негативно влияет на работу сервера. Если выполнение скрипта останавливается на точке, и пауза по какой-либо внешней причине занимает достаточно большой промежуток времени N, может произойти переполнение памяти и перегрузка сервера, и приложение будет работать некорректно. Для предотвращения таких ситуаций было решено ввести ограничение срока действия точек останова. В качестве максимального времени паузы исполнения скрипта на точке останова было решено выбрать период в три минуты, поскольку именно такой срок является максимальным для исполнения скриптов рабочих процессов.

Логика управления точками останова реализована в классе DebuggerMessagingAdapter, и осуществляется с помощью синтаксического анализа сообщений, проходящих через соединение.

Для решения данной задачи был выбран следующий метод – при создании точки останова необходимо запустить соответствующий ей таймер, который, по истечению трех минут запустит процесс удаления этой точки. Реализация данного подхода описана ниже.

При создании каждой новой точки останова в графическом интерфейсе, на стороне сервера приложения создается объект класса BreakpointWithLimitedLifeSpan, атрибутами которого является идентификатор точки и таймер, который инициализируется при создании объекта. Каждый объект помещается в хэш-таблицу класса управления (Приложение 9. BreakpointLifeSpanHandler.kt). По истечении срока таймера, по вебсокетному соединению отправляется сообщение об удалении точки останова, и, если исполнение скрипта в данный момент было остановлено на ней, о возобновлении исполнения программы. После этого объект BreakpointWithLimitedLifeSpan, соответствующий данной точке останова, удаляется из хэш-таблицы. В случае, если пользователь самостоятельно удалил точку в графическом интерфейсе до истечения срока действия таймера, соответствующей этой точке объект также удаляется из хэш-таблицы.

Однако в ходе реализации данного метода было замечено, что можно снизить потребление памяти и обойтись без создания хэш-таблицы класса управления. Для того, чтобы решить проблему нагрузки на сервер, не обязательно удалять точку останова по истечению срока таймера, достаточно продолжить исполнение скрипта, если оно стояло на паузе более трех минут. Для этого метода предложена нижеописанная реализация.

При паузе исполнения скрипта на некоторой точке останова, на стороне сервера приложения запускается таймер, единый для всех точек (в силу того, что мы придерживаемся фундаментального предположения о том, что при отладке исполнение скрипта не может остановится более чем на одной точке останова одновременно). В случае, если пользователь самостоятельно не продолжил исполнение программы в течение срока действия таймера, по его истечении по вебсокетному соединению отправляется сообщение о возобновлении исполнения скрипта рабочего процесса. После этого таймер обнуляется.

Таким образом, срок действия точек останова является автоматически конечным и контролируется приложением, что решает проблему потенциально излишней нагрузки на сервер.

Глава 5. Заключение

5.1. Результаты работы

По итогу проделанной работы были достигнуты следующие результаты:

1. Был осуществлен сбор информации и сравнительная оценка характеристик существующих протоколов отладки, на основании этого анализа был выбран оптимальный протокол – Chrome DevTools Protocol;
2. Инструмент отладки был реализован с учетом всех особенностей архитектуры приложения YouTrack;
3. Экспериментально был найден оптимальный подход к реализации механизма аутентификации и ограничении прав доступа при отладке;
4. Инструмент был интегрирован в сервер приложения YouTrack и может использовать всю необходимую логику приложения, однако при этом работает изолированно, используя свое собственное вебсокетное подключение;
5. Для минимизации влияния отладчика на стабильную работу сервера была введена система ограничения времени паузы на точках останова;
6. Была построена безопасная и достаточно простая в использовании и мониторинге инфраструктура отладчика.

5.2. Направления дальнейшей деятельности

Реализация возможности многопоточной отладки

На данный момент в системе существует искусственное ограничение – в одном экземпляре приложния YouTrack отлаживать скрипты может не более чем один пользователь. Так как выполняется удаленная отладка приложения, существует потенциальная ситуация, в которой механизм отладки на одном приложении будут использовать несколько пользователей одновременно. Для поддержки этой ситуации необходима реализация многопоточного обмена вебсокетными сообщениями между отладчиком и приложением. В качестве одного из подходов можно провести эксперименты по модификации системы обмена сообщениями на вебсокетном соединении.

Возможность задания пользователем максимального времени паузы исполнения скрипта

Для снижения нагрузки на сервер приложения в качестве максимального времени паузы исполнения скрипта на точке останова было решено выбрать период в три минуты, поскольку именно такой срок является максимальным для исполнения скриптов рабочих процессов. Однако после интервью с пользователями было выяснено, что для ряда задач им было бы удобнее иметь возможность самим выставлять желаемый промежуток времени в качестве максимального времени паузы.

Тестирование

Для подтверждения успешного прохождения всех пользовательских сценариев, включая граничные случаи, необходимо написать ряд юнит-тестов:

1. Тестирование прохождения пользователем первого этапа аутентификации с генерацией секретного токена, где пользователь является:
    - пользователем без прав на чтение всех и редактирование хотя бы одного скрипта автоматизации;
    - пользователем с правами на чтение всех и редактирование хотя бы одного скрипта автоматизации;
2. Тестирование прохождения второго этапа аутентификации с валидацией токена, записанного в адресе вебсокетного соединения протокола, по следующим сценариям:
    - токен не был использован, найден в хэш-таблице и доступ к отладке может быть разрешен;
    - токен был использован один раз, не найден в хэш-таблице и доступ к отладке не может быть разрешен;
    - токен не был использован, но не найден в хэш-таблице и доступ к отладке не может быть разрешен;
3. Тестирование корректной обработки ошибок попытки установления вебсокетного соединения при отсутствии сети Интернет;
4. Тестирование возобновления исполнения скрипта после того, как заданное время активности точки останова истекло. При корректном прохождении теста после

остановки выполнения скрипта автоматизации на любой точке останова по прошествии трех минут его исполнение возобновляется.

Список использованной литературы

1. Würthinger, T., Van De Vanter, M. L., Simon, D., Pnueli, A., Virbitskaite, I. & Voronkov, A., *Multi-level Virtual Machine Debugging Using the Java Platform Debugger Architecture*, Springer Berlin Heidelberg 2010, p. 401-412
2. Chen P., Fang Y., Mao B. & Xie L., JITDefender: *A Defense against JIT Spraying Attacks*, Future Challenges in Security and Privacy for Academia and Industry. SEC 2011. IFIP Advances in Information and Communication Technology, vol 354. Springer, Berlin, Heidelberg
3. Sharma, A.J., *WebKit Remote Debugging Protocol*, Developer Track WWW 2012, Lyon https://www2012.universite-lyon.fr/proceedings/nocompanion/DevTrack_Slides/041_Better_Web_Development_With_WebKit_Remote_Debugging.pdf
4. Hofer P., Stancu C., Jovanovic, V., Kessler, P., Wimmer, C., Wuerthinger, T., Pliss, O. & Woegerer, P., *Initialize Once, Start Fast: Application Initialization at Build Time*, Proceedings of the ACM on Programming Languages, 2019
5. Grimmer, M., Seaton, C., Schatz, R. & Mössenböck, W.H., *High-Performance Cross-Language Interoperability in a Multi-Language Runtime*, Proceedings of the 11th Dynamic Language Symposium, 2015
6. Duboscq, G., Würthinger, T. & Mössenböck, W.H., *Speculation without regret: reducing deoptimization meta-data in the Graal compiler,* Proceedings of the International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools, 2014
7. Ahmadzadeh, M., Elliman, D. & Higgins, C., Novice programmers: An analysis of patterns of debugging among novice computer science students. Inroads, 2005
8. Ducasse, M. & Emde, A. M., *A review of automated debugging systems: Knowledge, strategies and techniques*. Proceedings of the 10th international conference on software engineering. Singapore: IEEE Computer Society Press, 1988
9. Chrome DevTools Protocol, Google https://chromedevtools.github.io/devtools-protocol/
10. Java Debug Wire Protocol, Oracle https://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdwp-spec.html
11. Tutorial: Remote debug https://www.jetbrains.com/help/idea/tutorial-remote-debug.html#a0169e0

12. JetBrains, IntelijIDEA official documentation
https://www.jetbrains.com/help/idea/configuring-synchronization-with-a-remote-host.html
https://www.jetbrains.com/help/idea/debugging-javascript-in-chrome.html
https://www.jetbrains.com/help/idea/attaching-to-local-process.html
https://www.jetbrains.com/help/idea/run-debug-configuration-node-js-remote
13. Oracle, GraalVM Enterprise Edition 19 Guide
https://docs.oracle.com/en/graalvm/enterprise/19/guide/platform/implement-instrument.html
14. Grigorik, I., *Driving Google Chrome via WebSocket API*, 2012
https://docs.oracle.com/en/graalvm/enterprise/19/guide/platform/implement-instrument.html
15. Tudonu, M., *Debugging Smart Contracts with Truffle Debugger: A Practical Approach*
https://hackernoon.com/debugging-smart-contracts-with-truffle-debugger-a-practical-approach-f56bf0600736

Приложение

```
@TruffleInstrument.Registration(id = "workflowsInspector", name = "YouTrack Chrome Inspector",
                                            version = "0.1", services = TruffleObject.class)
public class YouTrackInspectorInstrument extends TruffleInstrument {
    private static final int DEFAULT_PORT = 9229;
    private static final HostAndPort DEFAULT_ADDRESS = new HostAndPort(null, DEFAULT_PORT);
    private Server server;
    private ConnectionWatcher connectionWatcher;


    static final OptionType<HostAndPort> ADDRESS_OR_BOOLEAN = new OptionType<>("[[host:]port]",
                                                (address) -> {
        return new HostAndPort(host, port);
    }, (Consumer<HostAndPort>) ((address) -> address.verify()));


    @com.oracle.truffle.api.Option(name = "", help = "Start the Chrome inspector on
                                                [[host:]port]. (default: <loopback
                                                address>:" + DEFAULT_PORT +
            ")", category = OptionCategory.USER, stability = OptionStability.STABLE) //
    static final OptionKey<HostAndPort> Inspect = new OptionKey<>(DEFAULT_ADDRESS,
                                                ADDRESS_OR_BOOLEAN);
 / ... /
    @Override
    protected void onCreate(Env env) {
        OptionValues options = env.getOptions();
        env.registerService(new Inspector(server != null ? server.getConnection() : null, new
                                                InspectorServerConnection.Open() {
            @Override
            @SuppressWarnings("all") // The parameters port and host should not be assigned
            public synchronized InspectorServerConnection open(int port, String host, boolean
                                                wait) {
                if (server != null && server.wss != null) {
                    return null;
                }
                HostAndPort hostAndPort = options.get(Inspect);
                connectionWatcher = new ConnectionWatcher();
                hostAndPort = new HostAndPort(host, port);
                try {
                    server = new Server(env, "Main Context", hostAndPort, false, false, wait,
                                                options.get(HideErrors),
                                                options.get(Internal),
                            options.get(Initialization), null, options.hasBeenSet(Secure),
                                                options.get(Secure), new
                                                KeyStoreOptions(options),
                                                options.get(SourcePath), connectionWatcher);
                } catch (IOException e) {
                    PrintWriter info = new PrintWriter(env.err());
```

```
                info.println(new InspectorIOException(hostAndPort.getHostPort(),
                                                      e).getLocalizedMessage());
                info.flush();
            }
            return server != null ? server.getConnection() : null;
        }
    }, new Supplier<InspectorExecutionContext>() {
        @Override
        public InspectorExecutionContext get() {
            if (server != null) {
                return server.getConnection().getExecutionContext();
            } else {
                return new InspectorExecutionContext("Main Context", options.get(Internal),
                                                     options.get(Initialization), env,
                                                     Collections.emptyList(), err);
            }
        }
    }));
}

@Override
protected OptionDescriptors getOptionDescriptors() {
    OptionDescriptors descriptors = new YouTrackInspectorInstrumentOptionDescriptors();
    return new OptionDescriptors() {
        @Override
        public OptionDescriptor get(String optionName) {
            return descriptors.get(optionName);
        }
     / ... /
}
```

Приложение 1. YouTrackInspectorInstrument.java

```
@GeneratedBy(YouTrackInspectorInstrument.class)
@Registration(id = "workflowsInspector", name = "YouTrack Workflows Chrome Inspector", version =
"0.1")
public final class YouTrackInstrumentProvider implements TruffleInstrument.Provider {


    @Override
    public String getInstrumentClassName() {
        return
"com.oracle.truffle.tools.chromeinspector.instrument.YouTrackInspectorInstrument";
    }


    @Override
    public TruffleInstrument create() {
        return new YouTrackInspectorInstrument();
    }


    @Override
    public Collection<String> getServicesClassNames() {
        return Arrays.asList("com.oracle.truffle.api.interop.TruffleObject");
    }
}
```

Приложение 2. YouTrackInstrumentProvider .java

```
open class WorkflowsDebuggerServlet : WebSocketServlet() {

    override fun configure(factory: WebSocketServletFactory) {
        factory.register(DebuggerMessagingAdapter::class.java)
    }
}
```

Приложение 3. WorkflowsDebuggerServlet.kt

```
/**
 * DebugServletFilter is required for redirection to json, which contains information about
chrome inspector.
 * Needed to add authentication to the connection of the debugger
 */
@WebFilter(urlPatterns = {"/json"}, filterName = "debugServletFilter")
@Scope(value="local", proxyMode= ScopedProxyMode.TARGET_CLASS)
public class DebugServletFilter implements Filter {
    private FilterConfig filterConfig;
    private static ArrayList<String> pages;  // хранилище страниц

    public DebugServletFilter() {
        if (pages == null)
            pages = new ArrayList<>();
    }
    @Override
    public void destroy() {
        filterConfig = null;
    }
    @Override
    public void init(FilterConfig fConfig) {
        filterConfig = fConfig;
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;
        String url = req.getRequestURI();

        if (url.equals("/json")) {
            HttpServletResponse httpResponse = (HttpServletResponse) response;
            httpResponse.sendRedirect("/api/workflowsinspector/json");
            return;
        }
        filterChain.doFilter(req, res);
    }
}
```

Приложение 4. DebugServletFilter.java

```
class WipConnection : WipRemoteVmConnection() {


    private var currentPageTitle: String? = null


    @Volatile
    private var connectionsData: ByteBuf? = null
    private var pageUrl: String? = null
    private var webSocketDebuggerUrl: String? = null
    private var title: String? = null
    private var type: String? = null
    private var id: String? = null


    override fun doOpen(result: AsyncPromise<WipVm>, address: InetSocketAddress, stopCondition:
Condition<Void>?) {
    / ... /
    override fun createChannelHandler(address: InetSocketAddress, vmResult:
AsyncPromise<WipVm>): ChannelHandler {
        return object : SimpleChannelInboundHandlerAdapter<FullHttpResponse>() {
            override fun channelActive(context: ChannelHandlerContext) {
                super.channelActive(context)
                try {
                    context.pipeline().remove(this)
                    connectToPage(context, address, connectionsData!!, vmResult)
                } catch (e: Throwable) {
                    handleExceptionOnGettingWebSockets(e, vmResult)
                }
            }


            @Suppress("OverridingDeprecatedMember")
            override fun exceptionCaught(context: ChannelHandlerContext, cause: Throwable) {
                vmResult.setError(cause)
                context.close()
            }
        }
    }


    fun getJsonInfo(connectionsJson: ByteBuf,
                    result: AsyncPromise<WipVm>) {


        if (!connectionsJson.isReadable) {
            result.setError(JSDebuggerBundle.message("error.websocket.malformed.message"))
            return
        }


        val reader = JsonReader(ByteBufInputStream(connectionsJson).reader())
```

```
        if (reader.peek() == JsonToken.BEGIN_ARRAY) {
            reader.beginArray()
        }
        while (reader.hasNext() && reader.peek() != JsonToken.END_DOCUMENT) {
            reader.beginObject()
            while (reader.hasNext()) {
                when (reader.nextName()) {
                    "url" -> pageUrl = reader.nextString()
                    "webSocketDebuggerUrl" -> webSocketDebuggerUrl = reader.nextString()
                    "id" -> id = reader.nextString()
                    else -> reader.skipValue()
                }
            }
            reader.endObject()
        }
    }

    override fun connectToPage(context: ChannelHandlerContext,
                               address: InetSocketAddress,
                               connectionsJson: ByteBuf,
                               result: AsyncPromise<WipVm>): Boolean {

        val debugMessageQueue = createDebugLogger("js.debugger.wip.log", debugLogSuffix ?: "")
        debugMessageQueue?.let { logger ->
            logger.add(connectionsJson, "IN")
            result.onError {
                logger.add("\"$it\"", "Error")
            }
        }
        val pageConnections = SmartList<PageConnection>()
        if (webSocketDebuggerUrl == null)
            result.setError("Please check your permissions to debug")
        pageConnections.add(PageConnection(pageUrl, title, type, webSocketDebuggerUrl, id,
address))
        return !processPageConnections(context, debugMessageQueue, pageConnections, result)
    }

    override fun processPageConnections(context: ChannelHandlerContext,
                                        debugMessageQueue: MessagingLogger?,
                                        pageConnections: List<PageConnection>,
                                        result: AsyncPromise<WipVm>): Boolean {
        val debuggablePages = SmartList<PageConnection>()

        for (p in pageConnections) {
```

```
            if (url == null) {
                debuggablePages.add(p)
            } else if (Urls.equals(url, Urls.newFromEncoded(p.url!!),
SystemInfo.isFileSystemCaseSensitive, true)) {
                connectDebugger(p, context, result, debugMessageQueue)
                return true
            }
        }

        if (url == null) {
            chooseDebuggee(debuggablePages, -1) { item, renderer ->
                renderer.append("${item.title} ${item.url?.let { "($it)" } ?: ""}",
                        if (item.webSocketDebuggerUrl == null)
SimpleTextAttributes.GRAYED_ATTRIBUTES else SimpleTextAttributes.REGULAR_ATTRIBUTES)
            }.onSuccess {
                val webSocketDebuggerUrl = it.webSocketDebuggerUrl
                if (webSocketDebuggerUrl == null) {

result.setError(JSDebuggerBundle.message("js.debug.another.debugger.attached"))
                    return@onSuccess
                }

                currentPageTitle = it.title
                connectDebugger(it, context, result, debugMessageQueue)
            }
                    .onError { result.setError(it) }
        } else {
            result.setError(JSDebuggerBundle.message("error.connection.no.page", url))
        }
        return true
    }

}
```

Приложение 5. WipConnection.kt

```
@GeneratedBy(YouTrackInspectorInstrument.class)
final class YouTrackInspectorInstrumentOptionDescriptors implements OptionDescriptors {
    YouTrackInspectorInstrumentOptionDescriptors() {
    }
    public OptionDescriptor get(String optionName) {
        byte var3 = -1;
        switch(optionName.hashCode()) {
            case 799505186:
                if (optionName.equals("workflowsInspector.Secure")) {
                    var3 = 10;
                }
                break;
            case 732761355:
                if (optionName.equals("workflowsInspector.SourcePath")) {
                    var3 = 11;
                }
                break;
           / ... /
            case 298520048:
                if (optionName.equals("workflowsInspector.Attach")) {
                    var3 = 1;
                }
        }
        switch(var3) {
            case 0:
                return OptionDescriptor.newBuilder(YouTrackInspectorInstrument.Inspect,
"workflowsInspector").deprecated(false).help("Start the Chrome inspector on [[host:]port].
(default: <loopback
address>:9229)").category(OptionCategory.USER).stability(OptionStability.STABLE).build();
            / ... /


            case 13:
                return OptionDescriptor.newBuilder(YouTrackInspectorInstrument.WaitAttached,
"workflowsInspector.WaitAttached").deprecated(false).help("Do not execute any source code until
inspector client is attached.
(default:false)").category(OptionCategory.USER).stability(OptionStability.STABLE).build();
            default:
                return null;
        }
    }
}
```

Приложение 6. YouTrackInspectorInstrumentOptionDescriptors.kt

```
open class ScriptingContextFactory(private val timeoutPropertyName: String =
WORKFLOW_TIMEOUT_PROPERTY_NAME) {


   / ... /
    val options by lazy(LazyThreadSafetyMode.PUBLICATION) {
        mapOf(
                "js.commonjs-require" to "true",
                "workflowsInspector" to "127.0.0.1:4242", // remote inspection
                "workflowsInspector.Suspend" to "false", // required to avoid pause on the first
line of the executable .js
                "js.experimental-foreign-object-prototype" to "true",
                "js.console" to "false",
                "workflowsInspector.Secure" to "false",
                "js.commonjs-require-cwd" to scriptsFileSystem.ROOT_FOLDER
        )
    }


    val engine by lazy(LazyThreadSafetyMode.PUBLICATION) {
        Engine.newBuilder()
                .allowExperimentalOptions(true)
                .options(options)
                .build()
    }
    private val contextBuilder by lazy(LazyThreadSafetyMode.PUBLICATION) {
        val hostAccess = HostAccess.newBuilder()
                .allowAllImplementations(true)
                .allowListAccess(true)
                .allowArrayAccess(true)
                .allowPublicAccess(true)
                .build()
        Context.newBuilder("js")
                .engine(engine)
                .allowExperimentalOptions(true)
                .allowIO(true)
                .allowHostAccess(hostAccess)
                .allowHostClassLookup(classShutter::visibleToScripts)
                .fileSystem(scriptsFileSystem)


    }


    / ... /
```

Приложение 7. ScriptingContextFactory.kt

```
/**
 * DebuggerMessagingAdapter responsible to handle connection, receive and forward data
 */
class DebuggerMessagingAdapter : WebSocketAdapter() {


    private val breakpointsHandler = getBean<BreakpointsLifeSpanHandler>()
    private val userTokenMap = getBean<MutableMap<XdUser, String>>("userTokenMap")


    // sort of session caching
    private val debuggerInstrumentSession = getBean<DebuggerInstrumentSession>()
    private val debugSessionContext = getBean<DebugSessionContext>()
    override fun onWebSocketConnect(session: Session) {


        val currentToken = (session as WebSocketSession).requestURI.path.split('/').last()
        // connect inspector only if token belongs to the certain YouTrack user otherwise debug
session is closed immediately
        if (userTokenMap.containsValue(currentToken)) {
            val currentYouTrackUser = userTokenMap.filterValues { it ==
currentToken }.keys.first()


            super.onWebSocketConnect(session)
            userTokenMap.remove(currentYouTrackUser)


            // todo add user login to logging
            securityLogger.debug("Token was used for the user ${currentYouTrackUser.xdId}" +
                    " and removed from the map. UserTokenMap size is ${userTokenMap.size}")


            debuggerInstrumentSession.debugSession =
InspectServerSession.create(debugSessionContext.context,
                    false, null)
            debuggerInstrumentSession.debugSession.open(object : MessageEndpoint {
                override fun sendText(message: String) {
                    if (session.isOpen) {
                        logger.debug("Workflow inspector send text: $message")
                        try {
                            session.remote.sendStringByFuture(message)
                            try {
                                val messageJson = JsonParser.parseString(message) as JsonObject
                                // when the breakpoint is hit, timer starts
                                breakpointsHandler.checkAndIfNeededStartTimer(messageJson)
                                // when the script execution is resumed, timer is canceled
                                breakpointsHandler.checkAndIfNeededCancelTimer(messageJson)


                            } catch (e: Exception) {
                                logger.debug("Fail to parse debug message: $message")
```

```
                            }


                        } catch (e: Exception) {
                            logger.error("Error when broadcasting message: $message")
                        }


                    }
                }

                / ... /
                override fun sendClose() {
                    logger.debug("Workflow inspector send close")
                    session.close()
                }
            })
        } else {
           securityLogger.debug("Auth token could not be used. UserTokenMap size is $
{userTokenMap.size}")
        }
    }

    override fun onWebSocketClose(statusCode: Int, reason: String?) {
        super.onWebSocketClose(statusCode, reason ?: "undefined reason")
        try {
            debuggerInstrumentSession.debugSession.sendClose()
        } catch (e: IOException) {
            logger.debug { "Inspector session is already closed" }
        }
        logger.info { "Inspector websocket closed" }


    }


    override fun onWebSocketText(message: String) {
        logger.debug("Workflow inspector on web socket text: $message")
        debuggerInstrumentSession.debugSession.sendText(message)
    }


}
```

Приложение 8. DebuggerMessagingAdapter.kt

```
@LocalScoped
@Component
class BreakpointsLifeSpanHandler {

    private val breakpointsTimer: Timer = getBean(Timer::class.java)
    private val debuggerInstrumentSession = getBean<DebuggerInstrumentSession>()
    private var expireTask = ExpireTask()

    // id = 1 indicates that this message was sent "artificially" to reduce server load
            private   val   resumeMessage   =   """{"id":1,"method":"Debugger.resume","params":
{"terminateOnResume":false}}"""

    internal inner class ExpireTask : TimerTask() {
        override fun run() {
            try {
                debuggerInstrumentSession.debugSession.sendText(resumeMessage)
            } catch (e: IOException) {
                logger.debug("The endpoint is already closed")
            }
        }
    }

    fun checkAndIfNeededStartTimer(messageJson: JsonObject) {
        try {
            val method = messageJson.get("method").toString()
            if (method == "\"Debugger.paused\"") {
                expireTask = ExpireTask()
                breakpointsTimer.schedule(expireTask, 10000) // Breakpoints life span is 10 sec
            }
        } catch (e: Exception) {
            logger.debug("Message is not related to starting the timer of breakpoint")
        }
    }

    fun checkAndIfNeededCancelTimer(messageJson: JsonObject) {
        try {
            val method = messageJson.get("method").toString()
            if (method == "\"Debugger.resumed\"") {
                expireTask.cancel()
               }
        } catch (e: Exception) {
            logger.debug("Message is not related to canceling the timer of breakpoint")
        }
    }
}
```

Приложение 9. BreakpointLifeSpanHandler.kt

```
/**
 * Chrome inspector as the truffle instrument.Production Version.
 */
@TruffleInstrument.Registration(id = "scriptsDebugger", name = "YouTrack Scripts
Debugger", version = "0.1", services = [TruffleObject::class])
class DebuggerInstrument : TruffleInstrument() {

    private var connectionWatcher: ConnectionWatcher? = null
    private val optionDescriptor = OptionDescriptor
            .newBuilder(OptionKey(true), "scriptsDebugger")
            .deprecated(false)
            .help("Enable the debugger for scripts")
            .category(USER).stability(STABLE).build()

    override fun onCreate(env: Env) {
        logger.debug("DebuggerInstrument is created")
        connectionWatcher = ConnectionWatcher()
        val server = DebuggerServer(env, "Main Context", emptyList<URI>())
        env.registerService(Inspector(env, server.connection,
                InspectorServerConnection.Open { _, _, _ ->
                    return@Open null
                }, {
            server.connection.executionContext
        }))
    }

    override fun onFinalize(env: Env) {
        if (connectionWatcher?.shouldWaitForClose() == true) {
            logger.debug("Waiting for the debugger to disconnect...")
            connectionWatcher?.waitForClose()
        }
    }

    override fun getOptionDescriptors() =
            object : OptionDescriptors {
                override fun iterator() = mutableListOf(optionDescriptor).iterator()
                override fun get(optionName: String) =
                            if (optionDescriptor.name == optionName) optionDescriptor
else null
            }
}
```

Приложение 10. DebuggerInstrument.kt