

СИСТЕМА ОБРАБОТКИ БИЗНЕС-ЛОГИКИ SERVER-SIDE ПРИЛОЖЕНИЯ НА GROOVY

Александр Шлянников
Digital Zone

Задача

- Возможность изменять бизнес-логику server-side Java EE приложения «на лету»:
 - С минимальными обращениями к разработчикам системы
 - Без перекомпиляции
 - Без shutdown/redeploy системы на сервере
 - С защитой от синтаксических и семантических ошибок

Применение

- Биллинговые системы:
 - Операторы связи
 - Такси
- Генерация разнообразных отчетов
- Пример:
 - «Клиенту, сделавшему 3 заказа в прошлом месяце и с днем рождения на этой неделе, сделать скидку в 10% после 15-й минуты поездки»

Типичные решения

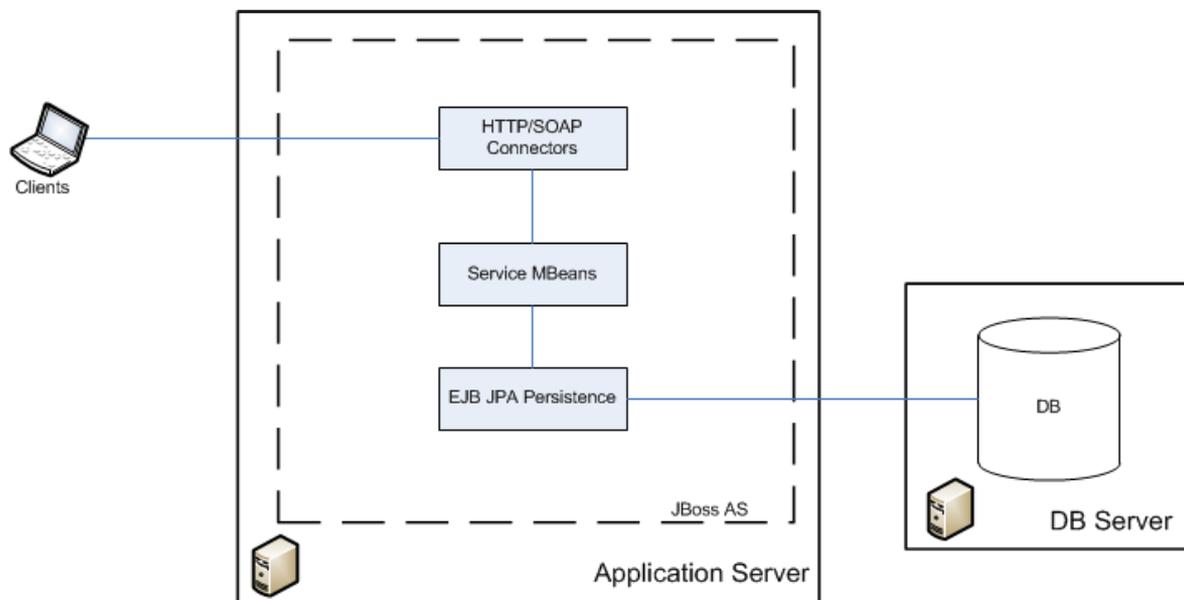
- Фиксированные параметры и настройки логики — недостаточно гибко
- Скриптинг:
 - JavaScript (Mozilla Rhino, <http://www.mozilla.org/rhino/>)
 - Groovy (<http://groovy.codehaus.org/>)

Groovy

- Dynamic language for the Java Virtual Machine:
 - Динамическая типизация
 - Удобный и краткий синтаксис работы с коллекциями, картами, массивами, строками
 - Возможность runtime-компиляции в JVM байт-код и работы с другим Java кодом и библиотеками

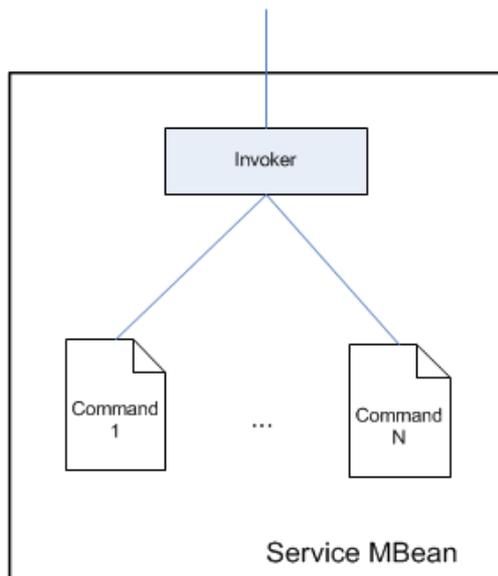
Архитектура

- Java EE – JBoss Application Server
- ORM – EJB JPA Persistence (Stateless & Entity Beans)
- Service MBeans
- HTTP/SOAP Client Connectors



Сервис команд

- Service MBean:
 - Invoker:
 - `Object invoke(String mapping, Object[] args)`
 - Commands:
 - `Object invoke(Object[] args)`



Оформление команд

- Команда: Groovy Script (класс)
- Runtime компиляция в JVM байт-код, создание объектов и хранение в памяти:
 - `GroovyClassLoader loader = new GroovyClassLoader();`
 - `Class groovyClass = loader.parseClass(content);`
 - `GroovyObject groovyObject = (GroovyObject) groovyClass.newInstance();`
- Файлы исходников команд расположены вне EAR/WAR/SAR-архивов
- Мониторинг изменений директории исходников через JBoss Deployer для runtime отслеживания изменений

Класс команды

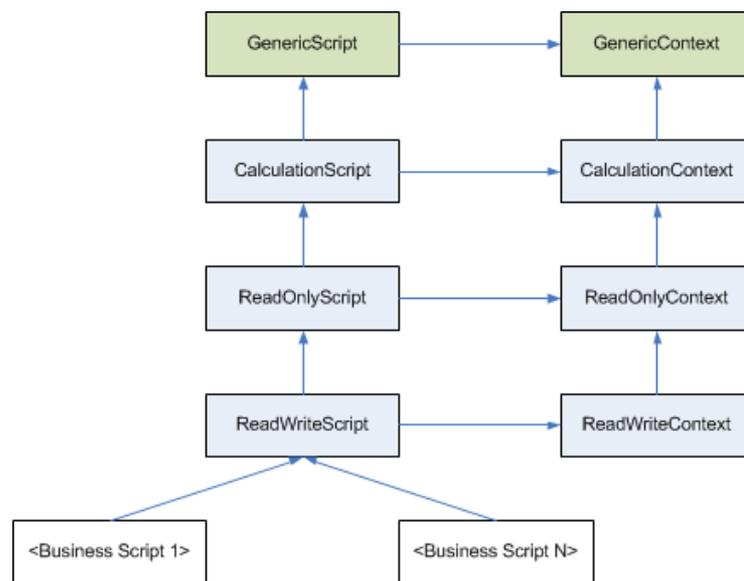
- Аннотация на класс – mapping команды:
 - `@ScriptMapping("/createOrder")`
- Имплементация Java интерфейса:
 - `public interface GenericScript {`
 - `void init(Object... args);`
 - `Object invoke(Object... args);`
 - `void onInterrupt(Object... args);`
 - `}`
- Хранение скомпилированных объектов в сервисе в виде ассоциативного массива [Mapping -> Object]
- Выполнение прямым вызовом метода `invoke` без использования Reflections:
 - `GenericScript s = scripts.get(mapping);`
 - `s.invoke(args);`

Базовый контекст выполнения скрипта

- Новое выполнение – новый объект (аналогично `HttpServletRequest`)
- УТИЛИТНЫЕ МЕТОДЫ:
 - `Object getAttribute(String key);`
 - `void setAttribute(String key, Object value);`
 - `Object invoke(String mapping, Object[] args);`
 - `void log(String message);`

Типы команд

- Разделение контекстов выполнения команд:
 - Calculation (базовый): без доступа к Persistence
 - Read-only: с доступом к Persistence на чтение
 - Read/Write: с доступом к Persistence на чтение/обновление



Организация доступа к данным

- EJB JPA Persistence:
 - Все сущности предметной области – @Entity
 - Утилитный Stateless Bean:
 - `public interface BaseDAO {`
 - `<T> List<T> getAll(Class<T> c);`
 - `<T> List<T> getEntitiesByKey(Class<T> c, String key, Object value);`
 - `<T> T getEntityById(Class<T> c, Object id);`
 - `<T> T createEntity(T entity);`
 - `void mergeEntity(Object entity);`
 - `void removeEntity(Class c, Object id);`
 - `}`
 - Методы Stateless Bean доступны через контекст скрипта

Организация доступа к данным

- Имплементация Stateless Bean, примеры:

- ```
<T> T createEntity(T entity) {
 • entityManager.persist(entity);
 • return entity;
• }
```

- ```
<T> List<T> getAll(Class<T> c) {  
    • Query query = entityManager.createQuery("select c from " +  
      c.getName() + " c");  
    • return query.getResultList();  
• }
```

Управление транзакцией

- Работа с транзакцией в Stateless Bean:
 - `@TransactionManagement(value = TransactionManagementType.CONTAINER)`
 - `@TransactionManagement(value = TransactionManagementType.BEAN)`
- При использовании CMT – аннотации на методах:
 - `@TransactionAttribute(TransactionAttributeType.REQUIRED)`
 - `@TransactionAttribute(TransactionAttributeType.SUPPORTS)`
 - ...
 - `sessionContext.setRollbackOnly(); // откат`
- В обоих случаях, нельзя:
 - `myStatelessBean.startTransaction();`
 - `doSomething();`
 - `myStatelessBean.commitTransaction();`

Управление транзакцией

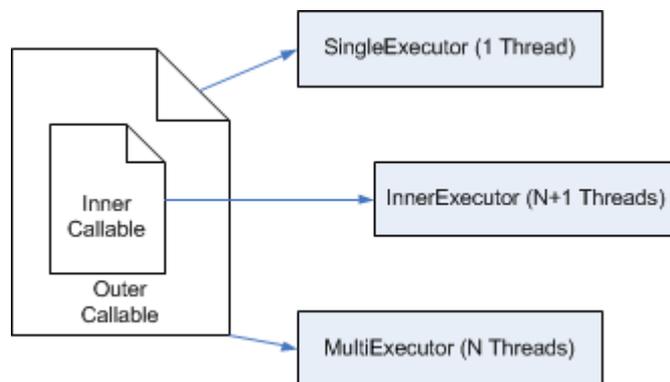
- Решение:
 - Специальные методы-обертки в Stateless Bean:
 - //для Read/Write контекста
 - `@TransactionAttribute(TransactionAttributeType.REQUIRED)`
 - `public Object wrapTransactionRequired(ScriptWrapper sw)`
 - //для Read-Only контекста
 - `@TransactionAttribute(TransactionAttributeType.SUPPORTS)`
 - `public Object wrapTransactionSupports(ScriptWrapper sw)`
 - Вызов метода `invoke` скрипта-команды и связывание с Stateless Bean – *внутри* методов `wrapTransactionRequired` и `wrapTransactionSupports`

Многопоточное исполнение

- Исполнение в очереди - `ExecutorService`:
 - `singleThreadExecutor`: один поток, контроль времени выполнения
 - `multiThreadExecutor`: несколько потоков, контроль времени выполнения
 - `debugThreadExecutor`: несколько потоков, без контроля времени выполнения
- Определение типа команды и таймаута выполнения в аннотации к классу скрипта:
 - `@Target(ElementType.TYPE)`
 - `@Retention(RetentionPolicy.RUNTIME)`
 - `public @interface ScriptMapping {`
 - `//...`
 - `long runTimeout() default -1;`
 - `ScriptThreadingType type() default ScriptThreadingType.MULTI;`
 - `}`

Контроль времени выполнения

- Два вложенных Callable на выполнение команды:
 - 1) Внутренний: запуск скрипта
 - 2) Внешний: контроль времени выполнения через `FutureTask.get(timeout)`
- Внутренний `ExecutorService` на $N+1$ поток

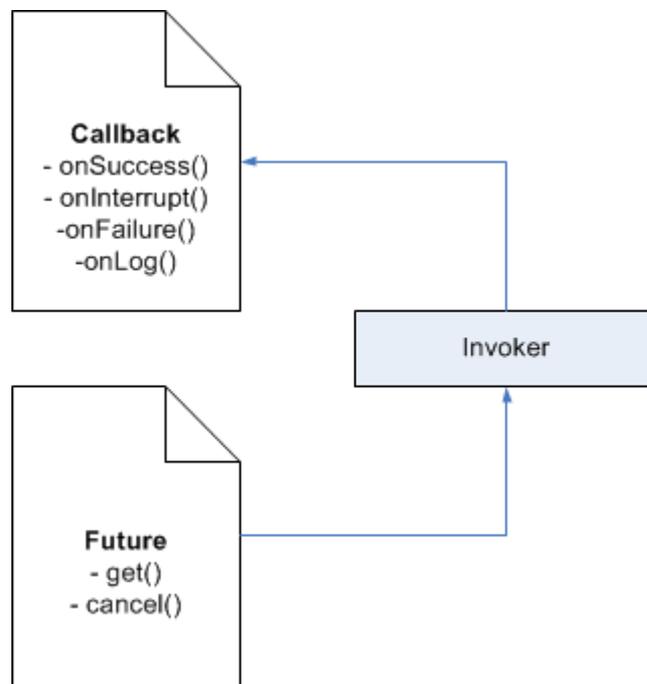


Контроль времени выполнения

- 1) `TimeoutException` в `FutureTask.get(timeout)`
- 2) Вызов метода `onInterrupt()` у скрипта команды для предупреждения о завершении
- 3) `sleep(timeout)`
- 4) `stop()` у потока
- 5) Запись в журнал ошибок

Асинхронный режим

- Вызывающий клиент имплементирует Callback для взаимодействия с командой во время выполнения, а получает Future:



Контроль ошибок

- 1) Проверка синтаксиса при компиляции:
 - `GroovyClassLoader loader = new GroovyClassLoader();`
 - `Class groovyClass = loader.parseClass(content);`
- - throws `CompilationFailedException` при синтаксической ошибке
- 2) Проверка времени исполнения по таймауту
- 3) При таймауте скрипта больше K раз – исключение из `Invoker`

Отладка

- Поддержка синтаксиса Groovy в IDE
- Удаленная отладка (JPDA) из IDE
- Выполнение в отдельном потоке без контроля таймаута

Интерфейс администрирования

- Create, Read, Update, Delete команд
- Версионность для отката изменений
- Мониторинг:
 - Количество команд в очереди
 - Exceptions
 - Отключенные команды

Пример

```
• @ScriptMapping(value = "/SetOrderToBoard", runTimeout = 10000L)
• class SetOrderToBoard extends ReadWriteScript {
•     def invoke(context, orderUuid, boardUuid) {
•         def success = false;
•         def order = context.findByKey("Order", "uuid", orderUuid);
•         def boards = context.findAll("Board");
•         for (board in boards) {
•             if (board.status == "free") {
•                 board.currentOrder = order;
•                 order.board = board;
•                 if (new Date().getTime() - order.creationTime > 10*60* 1000) {
•                     order.discount += 10;
•                 }
•                 context.update(board);
•                 context.update(order);
•                 success = true;
•                 break;
•             }
•         }
•         return success;
•     }
• }
```

Другие платформы

- Эквивалентное выполнение скриптов:
 - .NET - перенос в контекст отличающихся по синтаксису методов:
 - sqrt, pow, round, equal, etc

Выводы

- Разработанный сервис:
 - Глубокая настройка бизнес-логики приложения
 - Понятный юзерам язык и API
 - Работа с сущностями предметной области системы
 - Защита от ошибок
 - Возможность расширения на другие платформы

Спасибо за внимание!