Каталог Канонических Сервисов

Описание результатов тестового задания

В рамках тестового задания реализован сервис «Запрос текущего баланса абонента — getCustomerCurrentBalance» (п. 2.5.1.1.3.).

1. Архитектура

Согласно требованиям, необходимо было реализовать приложение, принимающее WS запрос, выполняющее обогащение запроса, маршрутизацию, отбор данных из источника и возврат ответа вызывающему web-сервису.

Использовались следующие предположения:

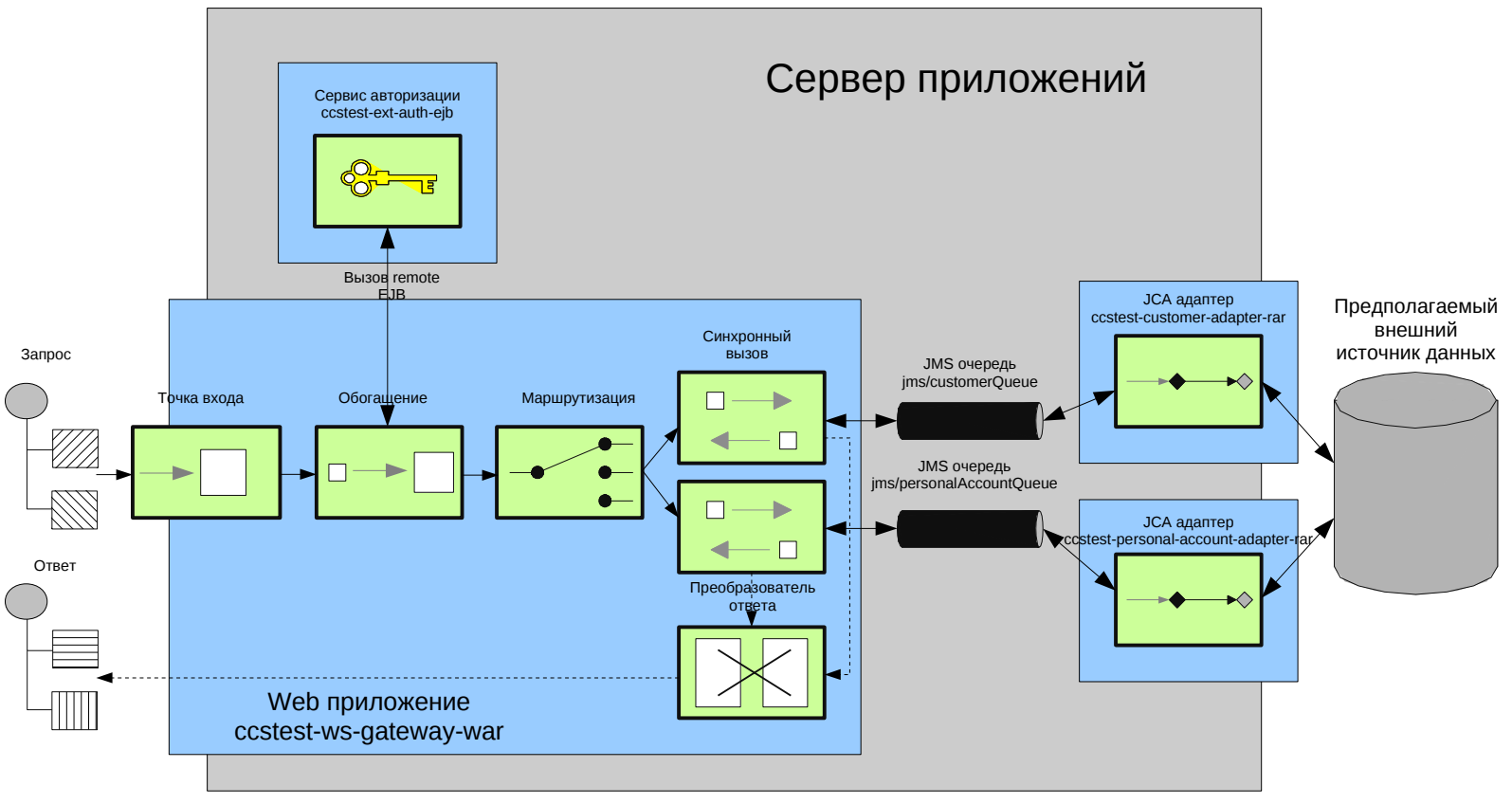
- входные интерфейсы должны быть «грубыми» (coarse grained) для облегчения реализации на стороне клиентов — фактически интерфейс определяется видом ответа и может принимать на вход различные виды запросов;
- приложение развертывается в JEE сервере приложений, работающем в кластере;
- входные сообщение поступают по протоколу SOAP поверх HTTP;
- точки обогащения (enrichment) запроса:
  - являются надежными (всегда доступны);
  - предоставляют ответ в течение короткого времени (всегда могут быть вызваны синхронно);
- источники данных:
  - могут быть временно недоступны;
  - могут обрабатывать запрос в течение длительного времени в синхронном либо асинхронном режиме.

Исходя из этих предположений была выбрана следующая архитектура:

- web-приложение (WAR) принимает запрос и выполняет обогащение (синхронные вызовы вне зависимости от транспорта) и маршрутизацию из одного потока (и соответственно одного JVM процесса в кластере);
- для каждого источника данных (в данном случае для каждого вида запроса) реализуется JCA (JEE Connector Architecture) адаптер (RAR); использование таких адаптеров позволяет явно отличать модули приема данных от модулей, работающих с источниками; кроме того для JCA адаптеров предоставляется автоматическая инициализация, доступ Work Manager API и т.д.;
- все общение между web-приложением и JCA адаптерами ведется через JMS (запросы от web-приложения могут передаваться JCA адаптерам, работающим в других JVM процессах);
- для JCA адаптеров все запросы являются асинхронными и их обработка может содержать сколь угодно сложную логику (например, асинхронный вызов нескольких источников, ведение очереди запросов в БД и т.п.);
- для реализации синхронных сервисов JMS вызов синхронизируется (in-out) в web-приложении (ожидание ответа во временную JMS очередь);
- для возврата ответа, имеющего большой объем, данные могут быть сохранены в БД, а их идентификатор передан по JMS (claim check).

2. Описание реализации

Схема работы:



Функционал приложения:

- **прием WS сообщения по HTTP (WS gateway)**: сообщение принимается с помощью JAX-WS и далее обрабатывается web-приложением в виде POJO;
- **обогащение (enrichment) сообщения**: для примера обогащения используется предполагаемый внешний сервис авторизации доступный как Remote Stateless EJB; полученный список ролей сохраняется в заголовке (header) сообщения (далее не используется);
- **маршрутизация сообщения на основе состава данных (content routing)**: маршрутизация выполняется на основе

заполненного поля «customerEntityId» либо «personalAccountEntityId»; перед маршрутизацией информация о составе сообщения сохраняется в заголовке (header);

- **использование JMS транспорта в синхронном режиме (in-out)**: сообщение сериализуется в XML и помещается в выбранную JMS очередь; вызывающая нить выполнения (thread) ожидает помещение ответа во временную JMS очередь (temporary queue); имя очереди для получения ответа передается в заголовке «`JMSReplyTo`»;
- **JCA адаптеры для доступа к источникам данных**: для доступа к предполагаемым источникам данных реализованы JCA адаптеры; каждый адаптер ожидает сообщений в отдельной JMS очереди; принимаемые сообщения десериализутся в POJO; в реализованном приложении адаптеры не обращаются к внешним источникам и возвращают заложенные в них данные (всегда одинаковые); ответ сериализуется в XML и передается во временную JMS очередь, указанную в заголовке «`JMSReplyTo`»;

Используемые технологии:

- для реализации обращений к ресурсам сервера приложений (JNDI), а также для реализации JCA адаптеров используется открытая библиотека Spring Framework (http://www.springsource.org/spring-framework , является рекомендуемой библиотекой для использования с WebLogic - http://www.oracle.com/technetwork/articles/entarch/spring-096279.html);
- для реализации бизнес-логики (получение, маршрутизация, передача) используется открытая библиотека Apache Camel (http://camel.apache.org/):
  - маршрут описывается виде XML с использованием Spring DSL;
  - для получения WS запросов используется компонент camel-cxf (использует JAX-WS реализацию от Apache CXF http://cxf.apache.org/);
  - для работы с JMS используется компонент camel-jms (для приема JMS сообщений использует Spring MessageListenerAdapter);
  - для описания логики обогащения (простой) используется встроенный в Camel язык выражений (expression language) Simple;
  - описания логики маршрутизации (сложной) выполнено на JavaScript (компонент camel-script, использует JSR223, не требует дополнительных библиотек);
- вся бизнес-логика описана через Camel, на Java написаны отдельные компоненты (утилиты для работы с JAXB, тестовые ответы и т.п.);

3. Ограничения тестового приложения

В целях сокращения времени выполнения тестового задания и упрощения развертывания следующие моменты **не** реализованы/**не** рассматривались (могут быть реализованы в дальнейшем):

- **внешние источники данных (эмулятор АСР):** данные для ответов заложены в коде компонентов JCA адаптеров;
- **HTTP безопасность и аутентификация**: предполагается использовать аутентификацию WS клиентов по X.509 сертификатам; может быть реализована как путем настройки сервера приложений, так и независимо от него для каждого web-приложения отдельно используя библиотеку Spring Security (http://www.springsource.org/spring-security , является рекомендуемым способом для WebLogic — http://docs.oracle.com/cd/E16764_01/web.1111/e14453/security.htm);
- **сохранение данных в БД:** доступ к БД может осуществляться с использованием Spring (получения DataSource либо EntityManager из JNDI) либо с использованием компонентов Camel: camel-jpa, camel-sql и т.д.;
- **XA транзакции:** транзакции могут быть подключены с использованием JtaTransactionManager (доступен из Spring, поддерживается в Camel) либо используя встроенный в Camel механизм глобальных транзакций;
- **использование WorkManager API:** сейчас Camel использует свои собственные thread-pool'ы; Camel может быть сконфигурирован для получения всех используемых thread'ов из WorkManager API;
- **обработка ошибок:** обработка ошибок выполнения для их возврата в виде SOAPFault не реализована, сейчас SOAPFault возвращается только при невозможности построения маршрута; требуется настройка Camel;
- **логирование:** настройка логирования для тестового приложения не проводилась, может потребоваться настройка логирования в сервере приложений;
- **конфигурирование:** конфигурирование приложения может осуществляться с использованием Spring Properties и Camel Properties; также можно добавить «горячее» конфигурирование с использованием JMX;
- **задание маршрутов в графическом редакторе:** для визуального задания и генерации маршрутов Camel могут быть использованы коммерческие IDE - Fuse IDE (http://fusesource.com/products/fuse-ide/) и Talend Open Studio (http://www.talend.com/products/open-studio-di.php);
- **WLST для описание логики маршрутизации:** для описания сложной логики маршрутизации может быть использован язык Python, который должен быть знаком пользователям WebLogic по WebLogic Scripting Tool (http://docs.oracle.com/cd/E15051_01/wls/docs103/config_scripting/using_WLST.html); сейчас используется JavaScript — так как он не требует дополнительных библиотек;
- **изменение маршрутов «по-горячему»:** изменение маршрутов без переразвертывания приложения может быть реализовано используя компонент camel-dynamic-router — позволяет задать произвольную логику маршрутизации, например, хранение маршрутов в БД и получение их из БД для каждого сообщения (с возможным кешированием и т.п.);
- **локальные вызовы EJB:** для вызова тестового сервиса авторизации используются remote вызовы, для

использования локальных вызовов требуется дополнительная настройка;

- **работа без сервера приложений:** реализованное приложение может быть собрано в виде standalone приложения (без изменения кода), используя embedded контейнер сервлетов (Tomcat либо Jetty) для работы JAX-WS; для JMS в таком случае можно использовать ActiveMQ (http://activemq.apache.org/);
- **интеграция с enterprise service bus (ESB):** Camel может быть использован совместно с ESB системой — в таком случае с использованием Camel описываются локальные маршруты и доступ к различным источникам/транспортам, а средствами ESB описываются глобальные маршруты с использованием Camel-модулей как «черных ящиков»;
- **работа в JBOSS, GlassFish и т.п.:** приложение не использует API и настройки, специфичные для WebLogic, но проверялось только в WebLogic 10.3.6.0.
- **работа с Java 5:** приложение требует для работы Java 6 (JRE), для работы с Java 5 необходимы изменения в библиотеках и пересборка.

4. Инструкции

Инструкция по установке:

- создать JMS Connection Factory с JNDI именем «jms/customerBalanceConnectionFactory»;
- создать JMS очереди с JNDI именами «jms/customerQueue» и «jms/personalAccountQueue»;
- развернуть приложение ccstest-dist-ear-1.0.ear;

Инструкция по сборке:

- установить Apache Maven 3;
- разархивировать ccstest.tar.gz;
- из папки ccstest выполнить «mvn clean install»;
- EAR будет собран по пути ccstest/ccstest-dist-ear/target/ccstest-dist-ear-1.0.ear.