Санкт-Петербургский государственный университет

БРОВКИН Александр Олегович

Выпускная квалификационная работа

Автоматизация бизнес-процесса на предприятии

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 Ренат Владимирович

Рецензент:
ведущий научный сотрудник кафедры прикладной кибернетики,
д.ф.-м.н. Киселева Мария Алексеевна

Санкт-Петербург
2021

Saint Petersburg State University

Aleksandr Brovkin

Bachelor’s Thesis

Business process automation

Scientific Supervisor:
Professor of Department of Applied Cybernetics,
Doctor of Science Renat Iuldashev

Reviewer:
Leading Researcher of Department of Applied Cybernetics,
Doctor of Science Mariia Kiseleva

Saint Petersburg
2021

Содержание

Введение 4

Описание предметной области 6

Постановка цели и задач 9

Глава 1. Обзор инструментов 11

1.1. BPMN 11

1.2. Camunda Platform 15

1.3. Camunda Modeler 17

Глава 2. Моделирование процесса 19

2.1. Декомпозиция 19

2.2. Процесс валидации запроса 19

2.3. Процесс выполнения запроса 22

2.4. Основной процесс обработки запроса 23

Глава 3. Автоматизация процесса 26

3.1. Обзор архитектуры 26

3.2. Интеграция с Jira 28

3.3. Формирование запроса 29

3.4. Запросы к СУБД 31

3.5. Интеграция с GitLab 33

3.6. Автоматизация процесса 34

Заключение 35

Список литературы 36

Приложение А. Модель процесса валидации запроса 39

Приложение Б. Модель процесса выполнения запроса 40

Приложение В. Модель основного процесса 41

Приложение Г. Исходный код интеграции с Jira 42

Приложение Д. Исходный код метода формирования запроса из заявки 45

Приложение Е. Исходный код интеграции с GitLab 48

Введение

В некоторой IT-компании, работающей с большими данными, имеется программное обеспечение, с определенной периодичностью выгружающее требуемое содержимое реляционной системы управления базами данных (СУБД) PostgreSQL в хранилище Hadoop Distributed File System. Параметры этой выгрузки описываются специальной конфигурацией, которую время от времени требуется обновлять, внося изменения. Обновление конфигурации выполняется по определенному алгоритму в соответствии с поступающим запросом.

Процесс формирования такой конфигурации, в совокупности со множеством других действий, входящих в алгоритм, является бизнес-процессом. Бизнес-процесс — устойчивая, целенаправленная совокупность взаимосвязанных видов деятельности, которая по определенной технологии преобразует входы в выходы, представляющие ценность для потребителя [1].

Бизнес-процессы позволяют получать воспроизводимый, повторяемый результат. Именно в этом заключается идеологическое отличие процесса от проекта, целью которого является какой-либо уникальный результат. Продукт процесса воспроизводим и повторяем [2]. Процесс имеет определенную последовательность. Подробно это означает, что, помимо определенного начала и определенного конца, существует также хронологическая и логическая последовательность.

На сегодняшний день существует множество средств для управления бизнес-процессами: событийные цепочки процессов (EPC-диаграммы) [3], сети Петри [4], диаграммы активности UML [5], IDEF0 и IDEF3 [6], а также BPMN [2] — наиболее молодая технология. Эти инструменты многократно сравнивали между собой [3; 7; 8], и в результате было показано, что BPMN имеет право на существование как нотация [9]. BPMN (*Business Process Model and Notation*) — нотация и модель бизнес-процессов. Разрабатывается организацией Object Management Group. Актуальная на текущий момент версия BPMN 2.0 была выпущена в январе 2011 года [10]. Главное преимущество BPMN перед многими другими нотациями заключается в том, что его стандарт охватывает не только графическую, но и исполняемую модель процесса [11],

в то время как другие нотации дают лишь техническое представление процессов. Кроме того, с 2013 года BPMN утвержден в качестве международного стандарта ISO [12].

Данная выпускная квалификационная работа посвящена исследованию возможностей моделирования исполняемых бизнес-процессов с использованием нотации BPMN 2.0, а также непосредственной реализации программного обеспечения, автоматизирующего отдельные шаги заданного процесса. В рамках работы BPMN используется для разработки модели процесса обновления конфигурации при помощи Camunda Modeler с целью дальнейшей автоматизации с использованием Camunda Platform [13].

Описание предметной области

Процесс выгрузки содержимого базы данных в хранилище, выполняемый специальным программным обеспечением, — это ETL-процесс [14]. ETL (*Extract, Transform, Load* — «извлечь, преобразовать, загрузить») состоит последовательно из:

1) извлечения данных из источника (в контексте данной работы источником является реляционная СУБД PostgreSQL);

2) трансформации данных для соответствия бизнес-модели;

3) их загрузки в хранилище (в данном случае — Hadoop).

Рассматриваемый ETL-процесс выполняется в соответствии с конфигурациями, представленными в формате JSON (*JavaScript Object Notation* — нотация объектов JavaScript) [15]. Пример такой конфигурации приведен в листинге 1.

```
{
  "schemaName": "schema",
  "tableName": "table",
  "columns": [
    {
      "name": "int_column",
      "dataType": "integer"
    },
    {
      "name": "str_column",
      "dataType": "varchar(32)"
    }
  ],
  "frequency": 60,
  "createSnapshots": true
}
```

Листинг 1: Пример конфигурации ETL-процесса

Таким образом, в каждой конфигурации процесса присутствуют следующие компоненты:

- название схемы в СУБД;
- название таблицы — источника данных;
- перечисление колонок таблицы, которые будут выгружаться: название колонки и соответствующий ей тип данных;
- частота выгрузки в минутах;
- логическая переменная-флаг, указывающая, создавать в хранилище слепки при загрузке либо перезаписывать данные каждый раз.

Каждая конфигурация представлена отдельным файлом, имеющем название вида «имя_схемы.имя_таблицы.json». Эти файлы хранятся в специальном репозитории в системе контроля версий GitLab [16].

Характерной особенностью этого ETL-процесса, без которой не возникло бы потребности в написании данной работы, является то, что регулярно требуется вносить изменения в конфигурации. Например, редактировать частоту или набор колонок, подлежащих выгрузке. В этом случае в системе управления задачами Jira [17] создается заявка, имеющая тип «Configuration Update» (запрос на обновление конфигурации). Заявка содержит в себе все поля, необходимые для формирования новой конфигурации. Поле, предназначенное для указания списка колонок, не является обязательным. Если перечисление отсутствует, при составлении конфигурации должны быть перечислены все колонки, имеющиеся в таблице.

Пример формы создания такой заявки представлен на рисунке 1a. На рисунке 1b — пример созданной заявки.

Обработка запроса на обновление конфигурации производится в виде последовательности операций, выполняемых вручную. Получая заявку соответствующего типа, сначала необходимо выполнить валидацию запроса, описываемого этой заявкой. Проверяется существование в базе данных указанной схемы и таблицы, а также набора колонок вместе с типами, если они перечислены. В случае последнего список колонок перед проверкой просматривается на наличие дубликатов, которые удаляются, если найдены. При отсутствии чего-либо (схемы в базе данных, таблицы в схеме, колонки

указанного типа в таблице) к заявке отправляется комментарий, содержащий информацию об ошибке, и ожидается внесение исправлений. По обновлении заявки валидация запроса выполняется заново. В результате успешной проверки об этом также отправляется комментарий.

Затем требуется выполнить запрос — составить конфигурацию в формате JSON в соответствии с примером, приведенным выше. Готовый файл конфигурации сохраняется в репозитории в новую ветку, названную по имени исходной заявки в Jira. В качестве сообщения фиксации изменений указывается ключ заявки и открывается запрос на слияние (merge request). Пользователь, запросивший обновление, проверяет, удовлетворяет ли подготовленная конфигурация всем требованиям, и, если все в порядке, отправляет изменения в ветку master (основная ветка репозитория). Если же он отклоняет изменения (закрывает запрос на слияние без слияния с master), то заявка должна быть закрыта этим пользователем либо сотрудником, обрабатывающим запрос.

Далее, в том случае, если слияние прошло успешно, запускается выполнение ETL-процесса путем вызова задачи «Release» в Jenkins и выполняется еще одна задача — обновление документации проекта в wiki-системе Confluence.

Для каждой таблицы, из которой производится выгрузка, в Confluence должна существовать страница (с названием вида «имя_схемы.имя_таблицы»), на которой перечислен список выгружаемых колонок с соответствующими им типами данных. Если на момент выполнения запроса необходимой страницы не существует, она создается, иначе — перезаписывается исходя из новой конфигурации.

Наконец, остается дождаться поступления уведомления о развертывании изменений. Уведомление приходит в виде комментария к исходной заявке в Jira. После этого заявка закрывается, на чем и завершается выполнение обработки запроса на обновление конфигурации.

(a) Форма создания новой заявки

(b) Созданная заявка

Рис. 1: Заявка с типом «Запрос на обновление конфигурации» в Jira

Постановка цели и задач

В данной работе решается проблема автоматизации описанного выше бизнес-процесса. В результате следующие действия должны выполняться автоматически, без участия пользователя-оператора: валидация запроса, формирование обновленной конфигурации, отправка созданного файла в новую ветку репозитория, создание запроса на слияние, вызов релиз-задачи, обновление документации и закрытие заявки.

Чтобы достигнуть поставленной цели, потребуется выполнить следующие задачи:

1. Смоделировать заданный бизнес-процесс с использованием BPMN.
2. Реализовать интеграции приложения с Jira, GitLab, Confluence.
3. Автоматизировать шаги процесса при помощи Camunda Platform.

Решение первой задачи предполагает преобразование текстового описания процесса в модель BPMN-процесса с учетом декомпозиции. Для решения

второй задачи потребуется при помощи библиотек, позволяющих взаимодействовать с API соответствующих сервисов, разработать методы, которые понадобятся для дальнейшей автоматизации. Третья задача будет логически поделена на две части:

a) программная реализация методов, которые должны вызываться при переходе управления к тому или иному действию процесса;

b) подготовка модели процесса к автоматизации: сопоставление каждому действию соответствующего метода, настройка условных переходов и передачи переменных внутрь подпроцессов и обратно.

Таким образом, конечным продуктом данной работы будут являться файлы моделей BPMN-процессов[1], настроенные для развертывания и запуска автоматизаций с использованием Camunda Platform, в совокупности с приложением[2], автоматизирующим процесс.

[1]Стандарт BPMN предполагает представление модели процесса в виде документа формата XML (*eXtensible Markup Language* — расширяемый язык разметки).

[2]Приложение — компьютерная программа, скомпилированная в исполняемый файл из исходного кода.

Глава 1. Обзор инструментов

В этой главе рассматривается стандарт BPMN 2.0 и основные элементы данной нотации. Также представлено используемые программное обеспечение: Camunda Platform и Camunda Modeler.

1.1 BPMN

Изначально BPMN обозначало Business Process Modeling Notation (нотация моделирования бизнес-процессов), которая предполагалась использоваться в качестве графического представления для Business Process Modeling Language (язык моделирования бизнес-процессов). BPML — язык, основанный на формате XML — не смог превзойти BPEL (язык исполнения бизнес-процессов), надмножеством для которого он являлся. В конечном итоге его разработка была прекращена в 2008 году.

Поначалу большой вклад в развитие внесла Business Process Management Initiative. Со временем разработку взяла на себя Object Management Group, выпустив в 2006 году BPMN 1.0. В версии 2.0, опубликованной в январе 2011 года, было добавлено множество нововведений: различные новых элементы нотации, единый формат представления BPMN-диаграмм. Также название было скорректировано на Business Process Model and Notation, то есть нотация **и** модель бизнес-процессов. Это отражало наиболее значимое нововведение в BPMN 2.0 — появление описания семантики выполнения процесса. Эта семантика, наряду с техническим моделированием процессов, обеспечивает техническую реализацию выполнения. Таким образом, диаграммы BPMN 2.0 могут выполняться напрямую с использованием движка системы управления бизнес-процессами [10]. По этой причине BPMN был выбран в качестве основного средства моделирования.

1.1.1 Графическая нотация

Стандарт BPMN 2.0 определяет множество графических элементов, которые позволяют с точностью описывать процессы. Элементы подразделяются на несколько категорий: события, действия, потоки, шлюзы, элементы данных.

Каждая из категорий имеет различные виды элементов. Далее вкратце будет приведено их описание.

События (Events) Используются для обозначения изменений свойств объекта, произошедших в какой-либо момент времени. Существует три типа событий: начальное, промежуточное и конечное. Они изображены на рисунке 2. Каждый процесс должен содержать минимум одно начальное и конечное событие. Название указывается под элементом.

Помимо перечисленных типов, события делятся на разные виды. Например, данной работе при моделировании процесса будет использоваться промежуточное событие получения сообщения.

Рис. 2: Типы событий

Действия (Activities) Представляют собой отдельные задачи в процессе. Название указывается внутри элемента. Ниже, на рисунке 3, представлены наиболее часто используемые при моделировании.

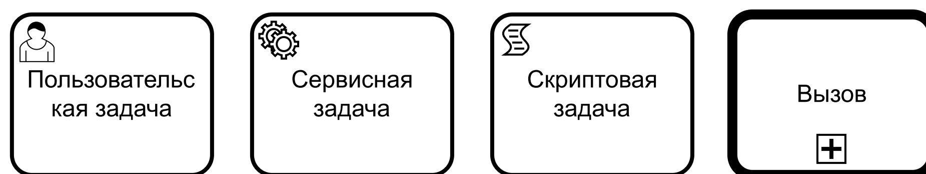


Рис. 3: Типы действий

- Пользовательская задача (User Task) предполагает участие человека.
- Сервисная задача (Service Task) — автоматическое выполнение какой-либо бизнес-логики. Например, вызов Java-кода.

- Скриптовая задача (Script Task) — выполнение скрипта.
- Вызов (Call Activity) — указание на процесс, являющийся внешним для текущего процесса, который требуется вызвать и передать ему управление.

Потоки (Flows) Отвечают за переход управления внутри процесса, соединяя между собой события и действия (см. рисунок 3). Обозначаются стрелками.

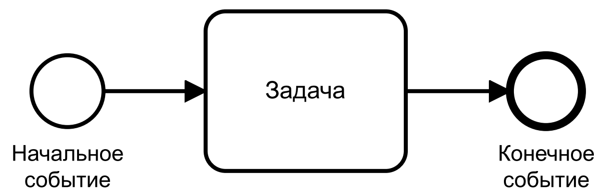
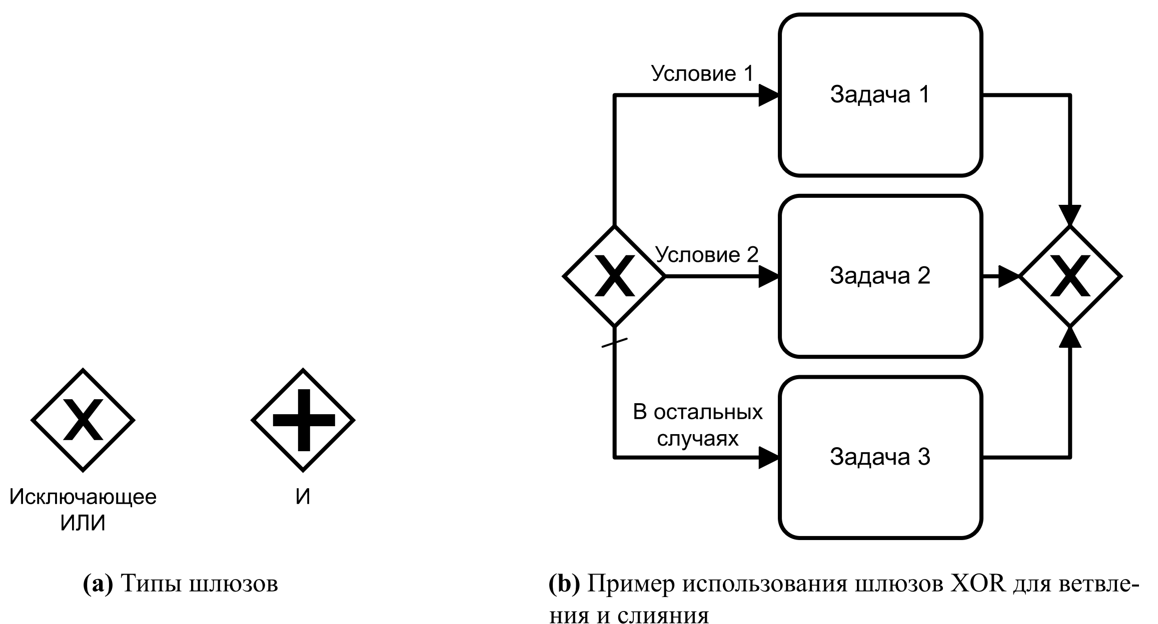


Рис. 4: Поток управления

Шлюзы (Gateways) Позволяют разветвляться потокам управления. На рисунке 5a изображены наиболее распространенные.



(a) Типы шлюзов

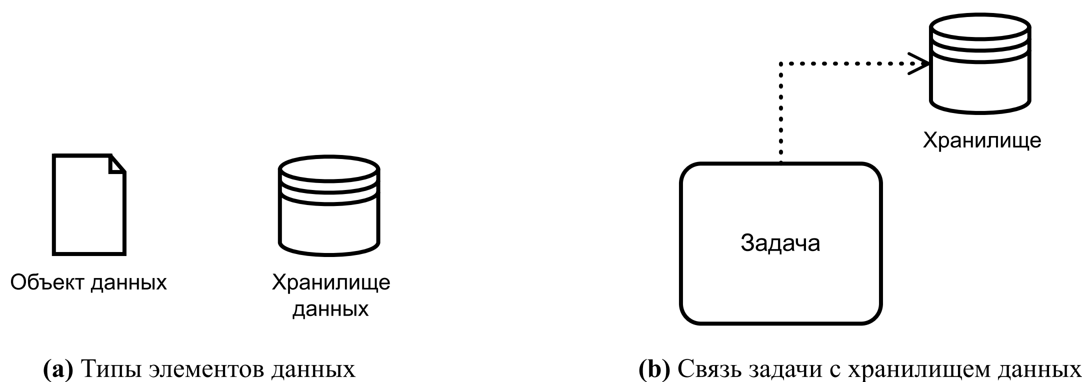
(b) Пример использования шлюзов XOR для ветвления и слияния

Рис. 5: Шлюзы

- Шлюз «Исключающее ИЛИ» (логическая операция XOR) передает управление по одному из исходящих потоков в зависимости от выполнения указанного условия. Если ни одно из условий не выполнено, используется поток по умолчанию (Default Flow), обозначаемый косой чертой в начале. Входящие потоки данный шлюз объединяет, получая управление от одного из них.

- Шлюз «И» (логическая операция AND) распределяет управление параллельно по всем исходящим потокам. Объединяет входящие, дожидаясь поступления от всех потоков

Элементы данных (Data Reference) Не влияют на выполнение процесса и используются только для наглядного изображения передачи информации. Представлены на рисунке 6a. С остальными элементами соединяются посредством ассоциативной связи (Association Flow), которая изображается стрелками, как на рисунке 6b.



(a) Типы элементов данных **(b)** Связь задачи с хранилищем данных

Рис. 6: Элементы данных

1.1.2 Happy Path

Happy Path (дословно «счастливый путь», далее обозначается как «позитивный сценарий»; термин произошел из тестирования программного обеспечения) — это стандартный вариант развития событий, который не содержит каких-либо проблем или ошибок. В контексте бизнес-процессов это сценарий, работающий наиболее простым и лучшим способом для бизнеса.

При моделировании рекомендуется сначала строить позитивный сценарий [18, Modeling Beyond the Happy Path]. Для этого требуется определить конечное событие, обозначающее желаемый результат процесса, затем найти подходящее начальное событие и последовательность действий, которые необходимы для достижения этого результата. После чего рассматриваются возможные ошибки и дополнительные сценарии: определяется нежелаемый результат — конечное событие, которое будет достигнуто в результате проблемы, — и прогнозируется конкретный шаг процесса, на котором может возникнуть ошибка. Кроме того, следует располагать задачи и события, составляющие позитивный сценарий, на одной прямой по центру диаграммы (см. рисунок 7).

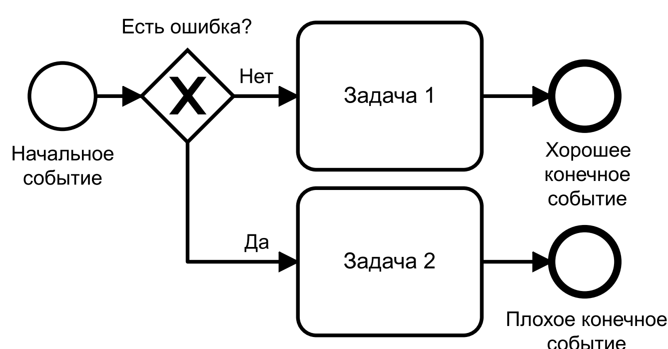


Рис. 7: Пример процесса с позитивным и негативным сценариями

1.2 Camunda Platform

Camunda Platform — это программный продукт с открытым исходным кодом [19], предназначенный для выполнения потоков работ и бизнес-процессов. В данной работе используется актуальная версия 7.15.0.

Camunda Platform предоставляет Java API (*Application Programming Interface* — интерфейс программирования приложений) и путем подключения соответствующих библиотек с легкостью интегрируется, например, в Spring приложение. Данный API позволяет приложению, разработанному с помощью Java, взаимодействовать с Camunda BPM. Платформа состоит из нескольких компонентов, необходимых для создания исполняемых процессов из BPMN-моделей и управления ими.

1.2.1 Process Engine

Процессный движок является главным элементом платформы Camunda. Его задача состоит в выполнении BPMN-моделей процессов. Движок интерпретирует модели в исполняемый код. При этом он следит за потоком управления процессов и определяет, какая задача должна быть выполнена далее. Путем Java API предоставляется возможность взаимодействовать с процессами, например, запускать их или управлять переменными. Это также используется приложениями, поставляемыми вместе с платформой: Cockpit и Tasklist. Устройство движка и взаимодействие с ним изображено на рисунке 8.

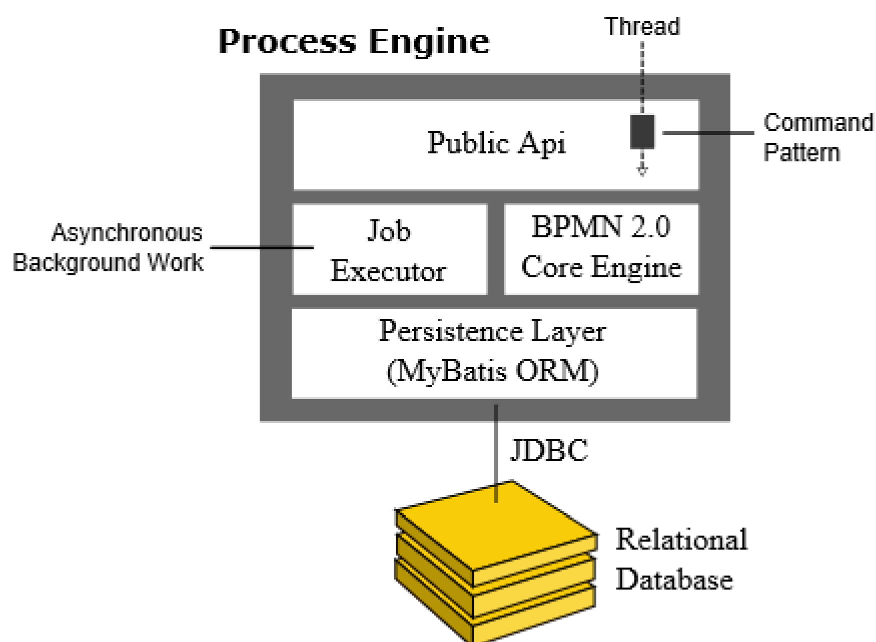


Рис. 8: Архитектура процессного движка [20, Introduction / Architecture]

1.2.2 Cockpit

Camunda Cockpit представляет собой веб-приложение, используемое для управления установленными в платформе определениями процессов и их экземплярами; контроля за переменными и местонахождением на диаграмме токенов отдельных экземпляров, а также ошибками, произошедшими во время выполнения.

1.2.3 Tasklist

Camunda Tasklist — еще одно веб-приложение в составе платформы. Предназначено для взаимодействия пользователя с процессом через назначенные ему задачи, а также для запуска экземпляров процессов с указанием начальных переменных. Приложение предоставляет обзор всех активных задач, помогая пользователю в их обработке.

1.3 Camunda Modeler

Для создания модели бизнес-процесса в данной работе используется приложение Camunda Modeler (актуальная версия 4.7.0). Оно не является компонентом платформы и устанавливается отдельно. Однако как часть экосистемы Camunda оно обеспечивает наилучшую совместимость при выполнении готовой модели процесса.

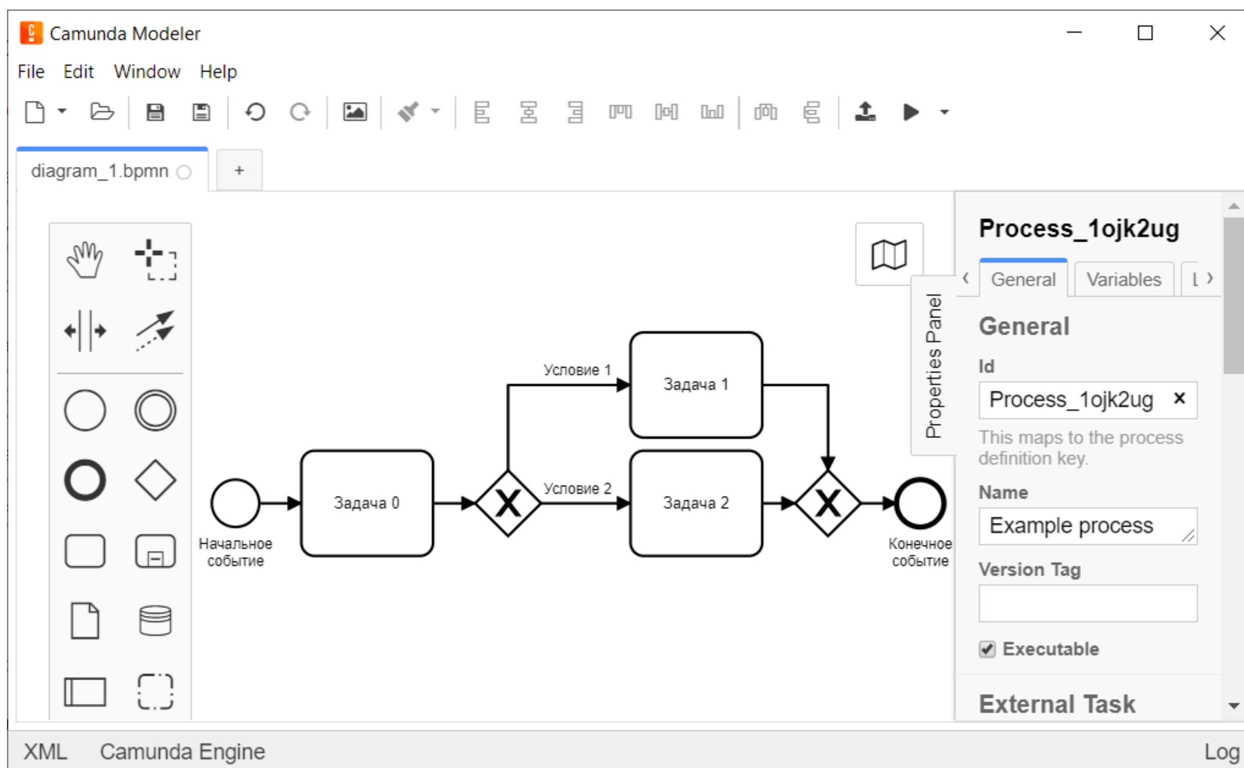


Рис. 9: Рабочее окно Camunda Modeler

Приложение имеет интуитивно понятный интерфейс (представлен на рисунке 9). Слева расположен набор всех доступных для моделирования эле-

ментов BPMN, которые можно размещать в рабочей области, составляя из них модель процесса. В верхней части окна — панель инструментов, предоставляющая широкий функционал: создание и сохранение новых диаграмм, импорт существующих, деплой готовых процессов в Camunda Platform и запуск с помощью Process Engine, экспорт диаграмм в виде изображений. Под панелью инструментов перечислены открытые на данный момент вкладки с диаграммами.

Помимо графического моделирования, Camunda Modeler позволяет редактировать все свойства, необходимые для технического выполнения процесса. Для этих целей справа находится панель свойств элементов. Также доступно редактирование XML-представления модели.

Кроме того, в Camunda Modeler предусмотрена автоматическая проверка модели на соответствие стандарту BPMN 2.0. Таким образом, при составлении диаграммы невозможно создать некорректную модель процесса, а при импорте существующей в логе приложения внизу окна появляется информация о наличии ошибок.

Глава 2. Моделирование процесса

В этой главе подробно рассматривается моделирование с использованием BPMN описанного ранее процесса обработки запроса на обновление конфигурации, который, исходя из целей и задач данной работы, в дальнейшем должен быть автоматизирован на основе разработанных моделей.

2.1 Декомпозиция

В первую очередь следует провести декомпозицию: выполнить переход от общего к частному. Метод декомпозиции позволяет заменить решение одной большой задачи путем объединения решений нескольких взаимосвязанных меньших задач. Таким образом, потребуется разделить исходный процесс на отдельные, более простые составляющие. Это приведет к повышению читаемости и наглядности диаграмм ввиду уменьшения их размеров, а также, к примеру, позволит повторно использовать действия, описанные в том или ином подпроцессе, многократно ссылаясь на него из различных процессов, не обязательно связанных между собой по смыслу [21]. В BPMN на подпроцесс можно ссылаться, используя задачу вызова, при этом сам подпроцесс по своей сути является процессом.

Составляющими основного процесса будут следующие подпроцессы:

1) валидации запроса;

2) выполнения запроса;

3) обновления документации.

Они, в свою очередь, состоят из множества различных задач. Рассмотрим отдельно каждый из подпроцессов.

2.2 Процесс валидации запроса

Выполнение валидации подразумевает контроль входных данных поступившего запроса. В процессе составляется текст комментария, который по окончании должен быть отправлен к заявке.

Сначала построим позитивный сценарий процесса (см. рисунок 10): запрошенные таблица и схема существуют, предоставлен список колонок, в котором нет дублирующихся или несуществующих в источнике колонок.

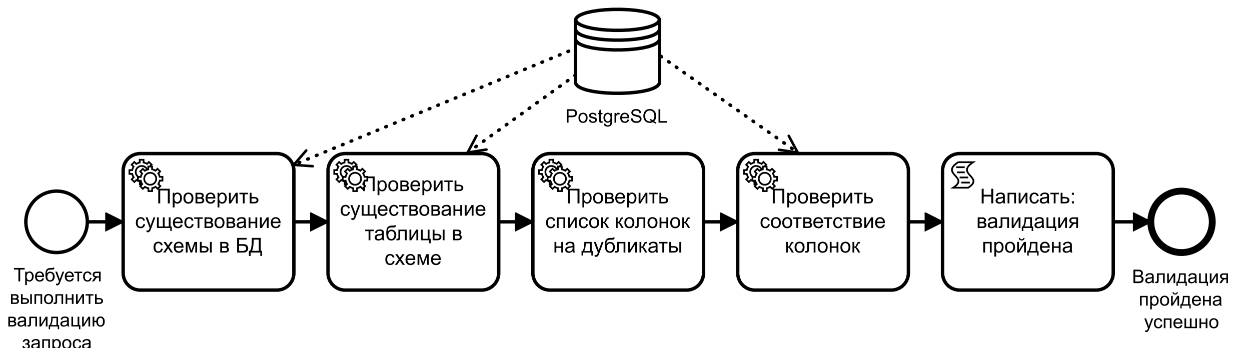


Рис. 10: Позитивный сценарий процесса валидации

Далее рассмотрим ситуации, когда имеются какие-либо ошибки, и добавим разделяющие шлюзы, позволяющие ответвляться от основного пути, по которому идет поток управления. На рисунке 11 представлена измененная модель, для компактности разделенная на две части (в рабочем варианте диаграммы все элементы лежат на одной прямой). Красным цветом выделены шлюзы, соответствующие возможным ошибкам. Фиолетовые шлюзы — не ошибки, но также нетипичные сценарии.

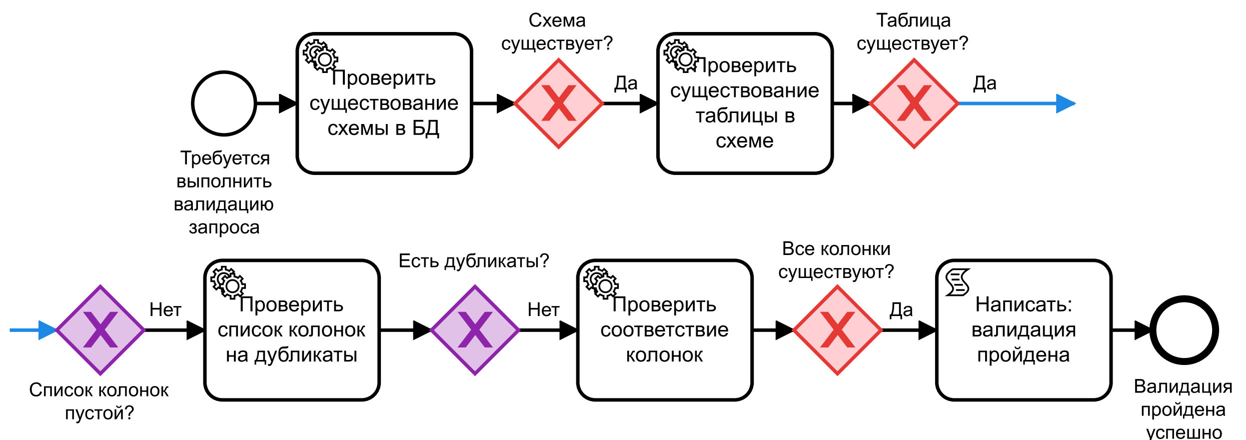


Рис. 11: Позитивный сценарий процесса валидации с добавлением шлюзов

Валидация может закончиться успехом, если все данные корректны, но в перечислении колонок присутствуют дубликаты. В этом случае дублика-

ты необходимо удалить, отметить это в тексте комментария и продолжить выполнение процесса. Данный сценарий изображен на рисунке 12.

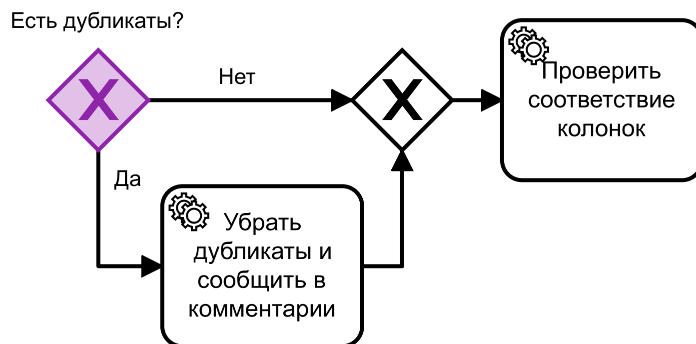


Рис. 12: Обработка дубликатов в процессе валидации

Кроме того, возможна ситуация, когда в запросе не приводится перечисление колонок. Это также не приводит к провалу валидации. При таком сценарии часть процесса, проверяющая указанные пользователем колонки, должна быть пропущена (см. рисунок 13).

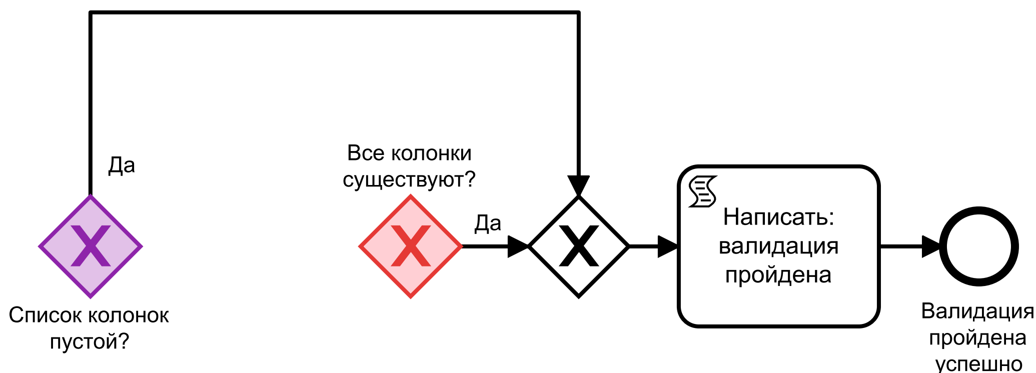


Рис. 13: Путь процесса валидации при отсутствии колонок

Теперь определим нежелательное конечное событие — неуспешную валидацию. Прежде чем до него добраться, требуется добавить в текст комментария информацию, сообщающую, в чем заключается ошибка. Для возможных ошибочных сценариев нужно определить соответствующие задачи — предоставление информации о конкретной ошибке. Ведут к этим задачам потоки управления, исходящие из добавленных ранее разделяющих шлюзов.

После индивидуальных действий эти потоки сливаются в один на объединяющем шлюзе, выполняется общее для всех ошибочных сценариев действие, затем процесс завершается. Обработка неблагоприятных ситуаций показана на рисунке 14.

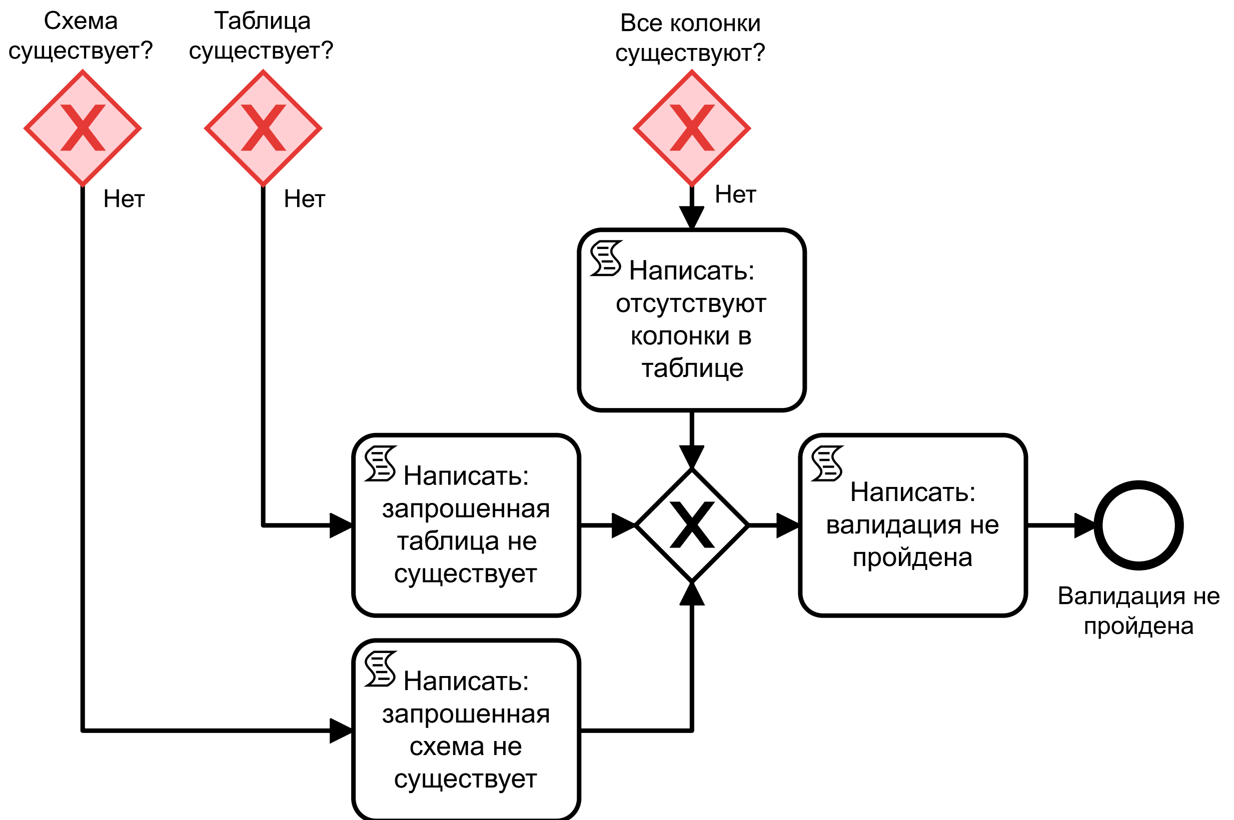


Рис. 14: Альтернативные пути процесса валидации

В приложении А приведен полный вариант модели процесса валидации.

2.3 Процесс выполнения запроса

Выполнение запроса на обновление — это генерирование новой конфигурации на основе поступивших данных, отправка файла в новую ветку репозитория и создание запроса на слияние. Для этого процесса также смоделируем позитивный сценарий. Он представлен на рисунке 15.

Рассматривая нестандартные сценарии, следует вспомнить, что указание перечисления колонок в заявке не является обязательным. В этом случае в конфигурации должны указываться все колонки, имеющиеся в таблице. Сле-

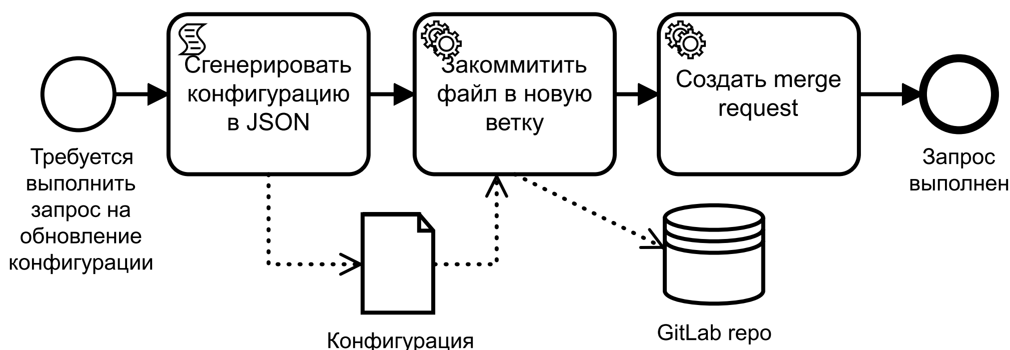


Рис. 15: Позитивный сценарий процесса выполнения

довательно, нужно добавить задачу получения списка колонок из источника данных (см. рисунок 16).

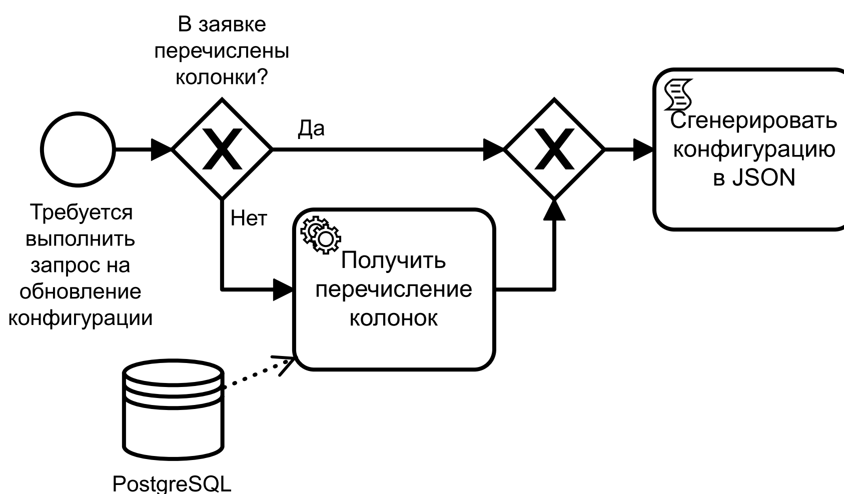


Рис. 16: Получение колонок в процессе выполнения

Полный вариант модели процесса представлен в приложении Б.

2.4 Основной процесс обработки запроса

Этот процесс является основным. Именно он должен начинать выполняться при поступлении новой заявки. Он состоит из смоделированных подпроцессов, а также некоторых других действий.

Опишем для него позитивный сценарий (см. рисунок 17). Сначала из исходных данных, поступивших с заявкой, формируется запрос. Далее идет

вызов первого подпроцесса — валидации. Отправляется комментарий, предполагаем, что валидация прошла успешно. В таком случае выполняется запрос — это второй подпроцесс.

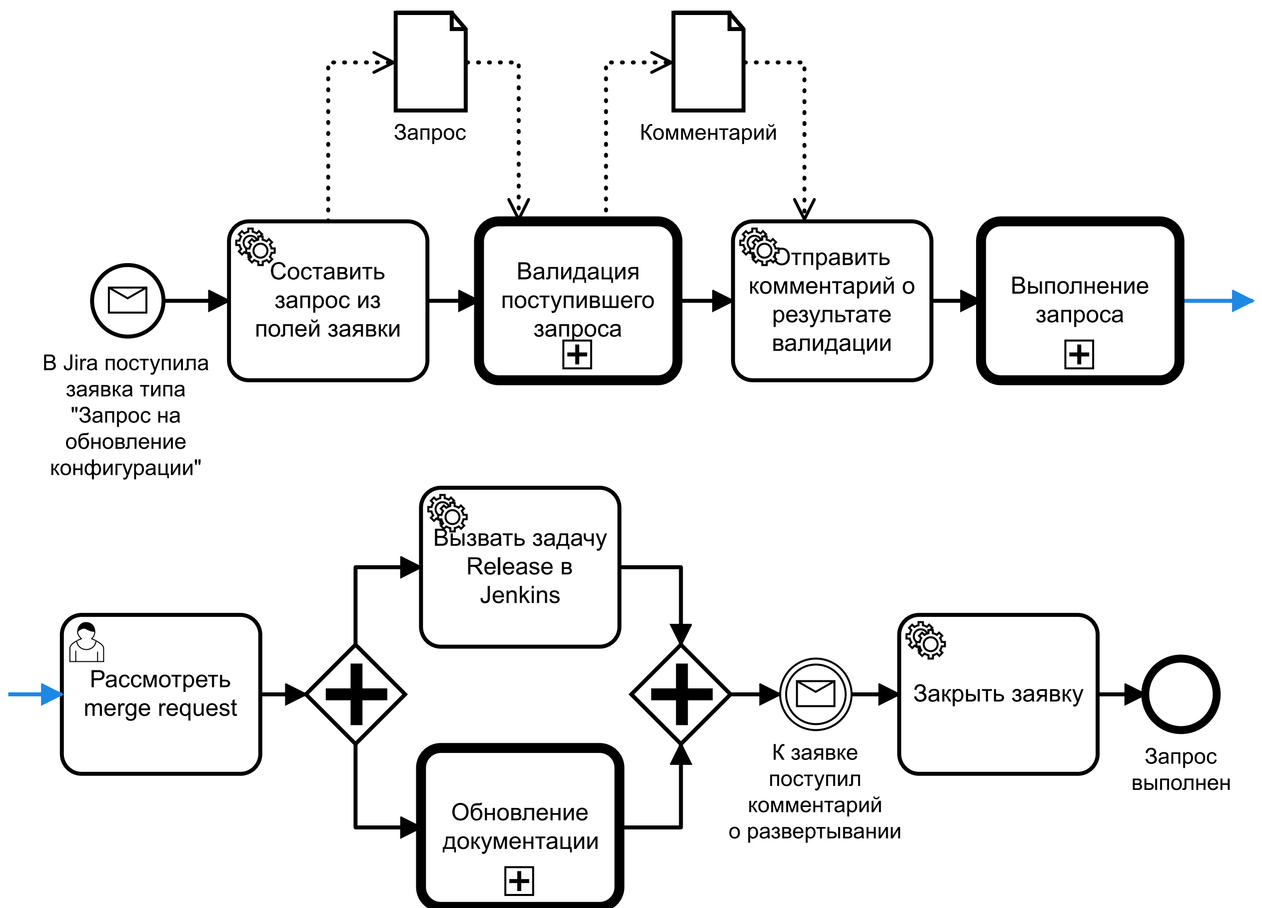
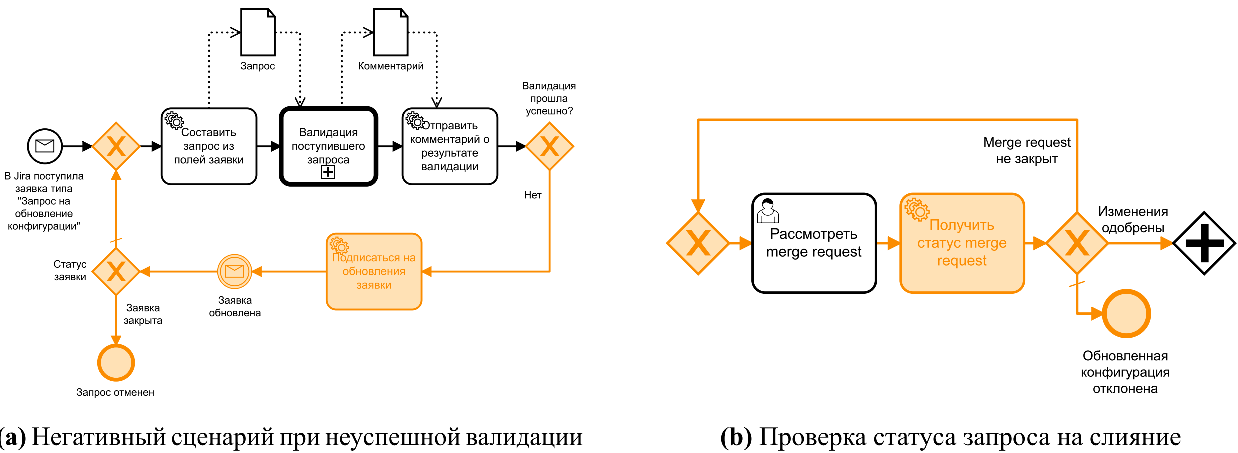


Рис. 17: Позитивный сценарий основного процесса

После его выполнения идет задача, которая заключается в проверке подготовленных изменений конфигурации. По выполнении проверки параллельно начинают выполняются два действия: вызов задачи «Release» и вызов подпроцесса, выполняющего подготовку обновленной конфигурации. Когда обе задачи выполнены, ожидается поступление комментария о развертывании изменений, после него закрывается заявка, и выполнение процесса завершается.

Теперь поочередно рассмотрим альтернативные сценарии процесса. Во-первых, это сценарий, возникающий в результате неуспешной валидации. Он изображен на рисунке 18а, действия, относящиеся к нему, выделены оранже-

вым цветом. В этом случае требуется подписаться на заявку и дожидаться ее обновления. Далее либо начать заново с составления запроса, либо завершить выполнение процесса.



(a) Негативный сценарий при неуспешной валидации

(b) Проверка статуса запроса на слияние

Рис. 18: Альтернативные сценарии основного процесса

Во-вторых, требуется контроль статуса запроса на слияние, когда пользователь отмечает задачу проверки изменений как выполненную. Если запрос на слияние принят, можно переходить к последующим задачам (это является частью позитивного сценария). Если изменения отклонены, то выполнение процесса завершается. Если же запрос все еще активен (изменения не приняты и не отклонены), пользователю требуется еще раз просмотреть его и принять решение о слиянии либо отклонении новой конфигурации.

На рисунке 18b изображены действия (подсвечены оранжевым), обрабатывающие различные сценарии, возникающие в зависимости от статуса запроса на слияние.

Целиком процесс приводится в приложении В.

Глава 3. Автоматизация процесса

Эта глава посвящена автоматизации процесса обработки запроса на обновление конфигурации. В ней описывается построение архитектуры приложения, рассматриваются интеграции со сторонними сервисами и некоторые ключевые моменты программного кода.

3.1 Обзор архитектуры

После построения модели процесса возникает вопрос, как сделать его исполняемым, а именно — как выполнять сторонний код. Автоматизация может пойти разными путями. Существует несколько способов развертывания Camunda Platform. Согласно официальным рекомендациям, лучше всего разрабатывать проект автоматизации процесса как Spring Boot приложение [18, Using a Greenfield Stack], подключая в качестве библиотек стартеры Camunda Spring Boot, предоставляющие API для взаимодействия приложения с процессным движком и веб-приложения Camunda.

Spring Boot [22] — это набор утилит, упрощающих процесс разработки, конфигурирования и развертки приложений на основе Spring Framework. При написании данной работы используется актуальная версия 2.4.5. Для разработки применяется Java 11 [23].

Опишем архитектуру проекта. За сборку и запуск отвечает система автоматической сборки Gradle [24]. Spring Boot приложение по умолчанию запускается на встроенном сервере Apache Tomcat [25, Chapter 3]. Само приложение содержит в себе:

- сервисы — различные интеграции, отвечающие за логику приложения;
- делегаты — классы, вызываемые во время выполнения процесса;
- простые Java-объекты — классы, моделирующие различные сущности;
- контроллеры — реализация REST (*Representational State Transfer* — передача состояния представления) API, позволяющая обращаться к приложению путем HTTP-запросов (*HyperText Transfer Protocol* — протокол передачи гипертекста);

- конфигурации — классы, отвечающие за установку соединения с СУБД или подготовку клиентов Jira и GitLab;

- Data Access Object (объект доступа к данным) — интерфейс к СУБД;

- встроенные ресурсы: файлы BPMN-моделей и HTML-форма (*HyperText Markup Language* — язык гипертекстовой разметки) пользовательской задачи процесса.

Интеграции с Jira и GitLab реализуются путем использования сторонних библиотек, существующих в открытом доступе [26; 27].

3.1.1 О взаимодействии с Jira

Функционал подключаемой библиотеки используется для, например, отправки к заявке комментария или ее закрытия. Однако он не позволяет подписываться на события, что необходимо для запуска выполнения процесса при создании новой заявки. Для этого случая Jira предоставляет возможность использовать технологию вебхуков [28, Chapter 9]. Вебхук — это инструмент уведомления клиента сервером о происходящих событиях посредством функции обратного вызова, предоставляемой клиентом. В данном случае это отправка HTTP-запроса по заранее установленному адресу. Существует одно ограничение: поддерживается только протокол HTTPS (*HTTP Secure* — безопасный HTTP) и порт 443. То есть необходима возможность шифрования трафика, идущего от Jira к приложению.

Таким образом, готовое приложение потребуется запускать на веб-сервере с дополнительно установленным SSL-сертификатом (*Secure Sockets Layer* — уровень защищенных сокетов). Для выпуска доверенного сертификата необходимо, чтобы серверу был назначен публичный статический IP-адрес (*Internet Protocol* — межсетевой протокол). В качестве сервера выбран Apache HTTP Server, установленный и запущенный на виртуальной машине с операционной системой Debian.

В результате была составлена схема, представленная на рисунке 19, которая демонстрирует архитектуру решения: приложение, содержащее код автоматизации и Camunda Platform, запускаемое на порте 8080 при помощи

Apache Tomcat; Apache HTTP Server, принимающий зашифрованный трафик через порт 443 и перенаправляющий его к приложению на порт 8080; виртуальная машина с Debian, предназначенная для развертывания.

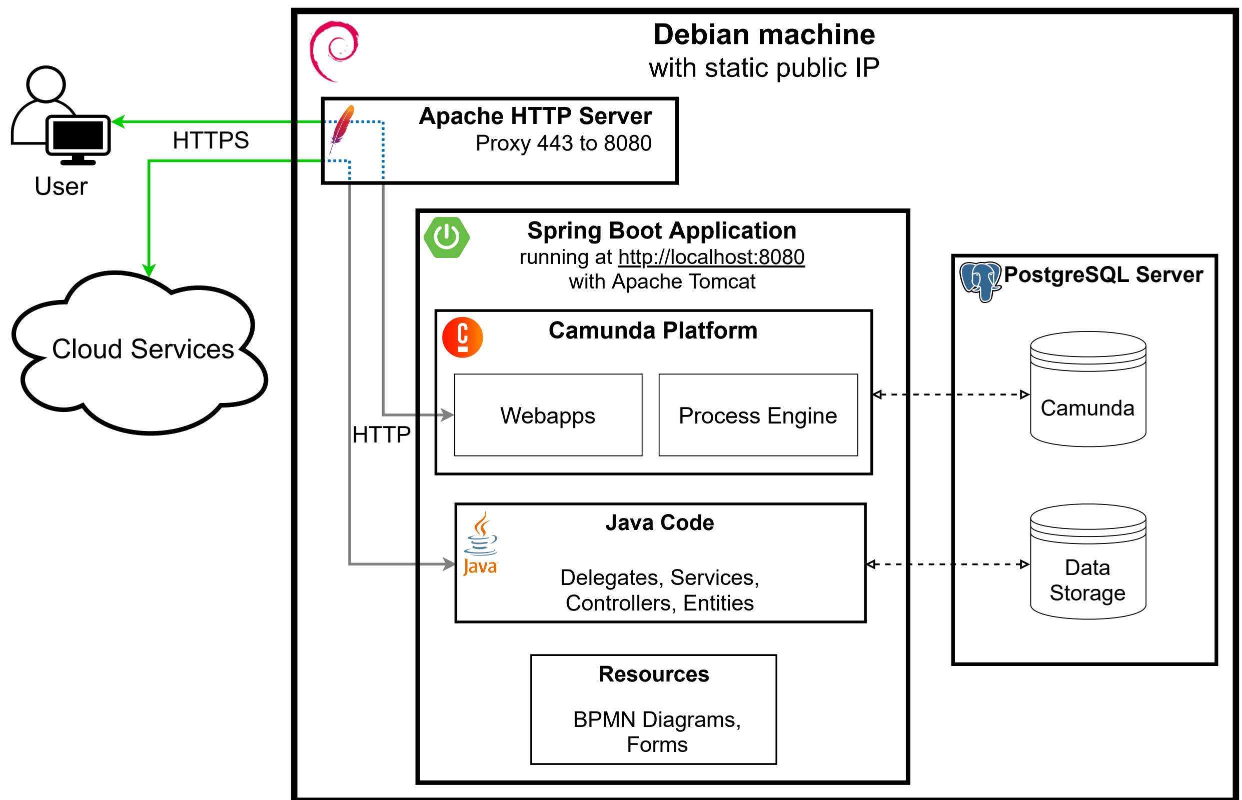


Рис. 19: Архитектура проекта автоматизации

3.2 Интеграция с Jira

Для реализации необходимых методов используется библиотека `jira-rest-java-client-core` [26] от Atlassian — разработчика Jira. Она предоставляет класс `IssueRestClient`, позволяющий взаимодействовать с заявками. Чтобы получить готовый для работы клиент, требуется реализовать класс конфигурации, создающий экземпляр клиента на основе адреса сервера Jira и данных для авторизации. Код представлен в листинге 2. Конфигурация помечается аннотацией `Configuration`. Аннотация `Value` используется для того, чтобы получить значение из свойств приложения и присвоить его соответствующей переменной.

```java
@Configuration
public class JiraRestClientConfig {
  @Value("${jira.url}")
  private String url;
  @Value("${jira.username}")
  private String username;
  @Value("${jira.api-token}")
  private String apiToken;

  @Bean
  public IssueRestClient issueRestClient() {
    JiraRestClient jiraRestClient =
      new AsynchronousJiraRestClientFactory()
        .createWithBasicHttpAuthentication(URI.create(url),
          username, apiToken);
    return jiraRestClient.getIssueClient();
  }
}
```

Листинг 2: Конфигурация REST-клиента Jira

Конкретная заявка описывается классом `Issue`. Для работы с заявками реализован класс `JiraIntegrationService` (см. приложение Г), содержащий методы для получения заявки по ключу, отправки к ней комментария, а также изменения статуса заявки.

3.3 Формирование запроса

Для получения полей из заявки был реализован перечисляемый тип, представленный в листинге 3.

Класс `ConfigurationUpdateRequest` из пакета `automation.domain` описывает запрос на обновление конфигурации. Для формирования запроса из заявки разработан метод, представленный в приложении Д. В нем, помимо прочего, обозначен список обязательных полей заявки, а также используются два паттерна регулярных выражений: один предназначен для проверки на корректность строки с перечислением колонок, второй позволяет получать

для каждой колонки отдельно имя и тип с длиной (см. листинг 4).

```
public enum ConfigurationUpdateFieldType {
  SCHEMA("Schema name"),
  TABLE("Table name"),
  COLUMNS("Set of columns"),
  FREQUENCY("Frequency"),
  CREATE_SNAPSHOTS("Create snapshots"),
  UNWANTED_FIELD("");

  private final String fieldTypeName;

  public static ConfigurationUpdateFieldType
  ↪  fromJiraName(String name) {
    for (ConfigurationUpdateFieldType type :
    ↪  ConfigurationUpdateFieldType.values()) {
      if (type.fieldTypeName.equalsIgnoreCase(name)) {
        return type;
      }
    }
    return UNWANTED_FIELD;
  }

  ConfigurationUpdateFieldType(String fieldTypeName) {
    this.fieldTypeName = fieldTypeName;
  }
}
```

Листинг 3: Перечисляемый тип с названиями полей Jira

```java
private static final List<ConfigurationUpdateFieldType>
↪  REQUIRED_FIELDS = List.of(SCHEMA, TABLE, FREQUENCY,
↪  CREATE_SNAPSHOTS);

private static final Pattern COLUMNS_FIELD_CORRECT_PATTERN =
  Pattern.compile(
    "^\\s*(?:\\w+\\s*:(?:\\w|\\s)*\\w+\\s*(?:|\\(\\s*
    ↪  \\d+\\s*\\)\\s*)(?:,\\s*(?!$)|$))+$"
  );
private static final Pattern COLUMNS_ITEMS_PATTERN =
  Pattern.compile(
    "(?<name>\\w+)\\s*:\\s*(?<type>(?:\\w\\s*)*\\w+)\\s*
    ↪  (?:\\(\\s*(?<length>\\d+)\\s*\\))?"
  );
```

Листинг 4: Регулярные выражения для проверки и обработки поля с колонками

3.4 Запросы к СУБД

Необходимо сконфигурировать два источника данных: первый — это база данных, необходимая для работы Camunda Platform, второй — непосредственно хранилище, из которого производится выгрузка. Класс конфигурации представлен в листинге 5.

```java
@Configuration
public class DataSourceConfig {
  @Bean("camundaBpmDataSource")
  @ConfigurationProperties("datasource.camunda")
  public DataSource camundaBpmDataSource() {
    return DataSourceBuilder.create().type(HikariDataSource
    ↪  .class).build();
  }

  @Bean("storageDataSource")
  @Primary
  @ConfigurationProperties("datasource.storage")
  public DataSource storageDataSource() {
    return DataSourceBuilder.create().type(HikariDataSource
    ↪  .class).build();
```

```
  }
}
```

Листинг 5: Конфигурация источников данных

Листинг 6 демонстрирует интерфейс, предназначенный для отправки запросов к СУБД. Описаны методы для проверки существования схемы, существования таблицы в схеме и получения описания колонок для указанной таблицы.

```
@Mapper
public interface DatabaseDao {

  @Select(
    "SELECT exists(SELECT FROM information_schema.schemata
    ↪  WHERE schema_name = #{schemaName})"
  )
  boolean schemaExists(String schemaName);

  @Select(
    "SELECT exists(SELECT FROM information_schema.tables
    ↪  WHERE table_schema = #{schemaName} AND table_name =
    ↪  #{tableName})"
  )
  boolean tableExists(String schemaName, String tableName);

  @Select(
    "SELECT column_name as columnName, data_type as dataType,
    ↪  character_maximum_length as charMaxLen FROM
    ↪  information_schema.columns WHERE table_schema =
    ↪  #{schemaName} AND table_name = #{tableName}"
  )
  Set<Column> tableDescription(String schemaName, String
  ↪  tableName);
}
```

Листинг 6: Интерфейс взаимодействия с СУБД

3.5 Интеграция с GitLab

Для взаимодействия используется библиотека `gitlab4j-api` [27]. Первоначально нужно настроить конфигурацию классов `GitLabApi` и `GitLabProperties`, как указано в листинге 7.

```
@Configuration
public class GitLabApiConfig {

  @Value("${gitlab.host-url}")
  private String hostUrl;
  @Value("${gitlab.access-token}")
  private String accessToken;
  @Value("${gitlab.repo-path}")
  private String repoPath;
  @Value("${gitlab.start-branch}")
  private String startBranch;

  @Bean
  public GitLabProperties gitLabProperties() {
    return new GitLabProperties(hostUrl, accessToken,
    ↪  repoPath, startBranch);
  }

  @Bean
  public GitLabApi gitLabApi() {
    return new GitLabApi(gitLabProperties().getHostUrl(),
    ↪  gitLabProperties().getAccessToken());
  }
}
```

Листинг 7: Конфигурация GitLab API

Реализованы методы, выполняющие загрузку файла в репозиторий (создание или обновление в зависимости от существования файла), создание запроса на слияние, проверку статуса запроса на слияние и преобразование строки в корректное название ветки. Исходный код интеграции находится в приложении Е.

3.6 Автоматизация процесса

Подавляющее большинство задач в процессе являются сервисными задачами. Вызов Java-кода из сервисной задачи возможен при помощи классов-делегатов, которые реализуют интерфейс `JavaDelegate` с единственным методом `execute`, принимающим в качестве параметра объект класса `DelegateExecution` [20, Reference / Service Task]. Этот объект описывает текущее состояние выполнения экземпляра процесса и позволяет, к примеру, управлять переменными процесса. Каждый класс-делегат должен быть помечен аннотацией `Component`. Делегаты находятся в пакете `automation.delegate`. Листинг 8 демонстрирует один из множества таких классов, реализованных для автоматизации процесса, в качестве примера.

```
@Component("buildRequestDelegate")
public class BuildRequestDelegate implements JavaDelegate {
  @Autowired
  private JiraIntegrationService jiraIntegrationService;

  @Override
  public void execute(DelegateExecution execution) {
    ProcessVariables variables = new
    ↪   ProcessVariables(execution);
    String issueKey = variables.getIssueKey();

    jiraIntegrationService.setIssueInProgress(issueKey);
    ConfigurationUpdateRequest request =
    ↪   ConfigurationUpdateRequest.of(jiraIntegrationService⌋
    ↪   .getIssue(issueKey));

    variables.setRequest(request);
    variables.setValidationStatus(new ValidationStatus());
  }
}
```

Листинг 8: Пример делегата

Исходный код разработанного приложения представлен в репозитории [29].

Заключение

В результате выполнения данной выпускной квалификационной работы была выполнена поставленная цель — автоматизирован заданный бизнес-процесс обработки запроса на обновление конфигурации выгрузки из СУБД. Процесс был смоделирован в соответствии с BPMN, а затем автоматизирован при помощи платформы автоматизации бизнес-процессов Camunda Platform. Разработано Spring Boot приложение, отвечающее за автоматизацию. При этом также реализованы интеграции приложения с системой управления заявками Jira и системой контроля версий GitLab.

По итогу разработки приложения началось его внедрение на предприятии в тестовом режиме. Результаты показывают, что автоматизация позволяет существенно оптимизировать работу предприятия. До внедрения прохождение такого бизнес-процесса с момента создания заявки занимало в среднем до одного рабочего дня. Автоматизация сократила время обработки запроса до нескольких секунд. Остался только один шаг процесса, требующий непосредственного участия пользователя, — проверка изменений и закрытие запроса на слияние. По приблизительным оценкам сотрудников после внедрения трудозатраты сократились на 90%. Кроме того, делегирование приложению заданий, требующих внимательности, позволяет избежать ошибок, связанных с человеческим фактором, что особенно актуально при формировании конфигураций, в которых перечисляется более пяти колонок.

Список литературы

1. *Репин В. В.* Моделирование бизнес-процессов в нотации BPMN. Пособие для начинающих. Часть I. — Издательские решения, 2019. — 84 с. — ISBN 978-5-44966989-6.

2. *Фёдоров И. Г.* Моделирование бизнес-процессов в нотации BPMN 2.0. — Москва : МЭСИ, 2013. — 255 с. — ISBN 978-5-77640772-7.

3. *Фёдоров И. Г.* Сравнительный анализ нотаций моделирования бизнес-процессов // Открытые Системы. СУБД. — 2011. — 8 (174). — С. 28—32.

4. *Питерсон Д.* Теория сетей Петри и моделирование систем / пер. с англ. М. В. Горбатовой, Т. В. Л., Ч. В. Н. — Москва : Мир, 1984. — 264 с. — ISBN 978-5-94074-334-7.

5. *Буч Г.*, *Рамбо Д.*, *Якобсон И.* Язык UML. Руководство пользователя / пер. с англ. Н. Мухина. — ДМК Пресс, 2006. — 496 с. — ISBN 978-5-94074-334-7.

6. *Noran O.* UML vs IDEF: An Ontology-oriented Comparative Study in View of Business Modelling // Proceedings of the 6th International Conference on Enterprise Information Systems. — 2005. — С. 674—682.

7. *Geambaşu C. V.* BPMN vs. UML Activity Diagram for Business Process Modeling // Accounting and Management Information Systems. — 2012. — Т. 11, № 4. — С. 637—651.

8. Анализ графических нотаций для имитационного моделирования бизнес-процессов предприятия / О. П. Аксенова [и др.] // Современные проблемы науки и образования. — 2013. — № 4. — С. 43. — ISSN 2070-7428.

9. *Белайчук А. А.* Главное преимущество BPMN // Открытые Системы. СУБД. — 2012. — 8 (184). — С. 61—62.

10. Business Process Model and Notation / Object Management Group. — Вер. 2.0. — 2011. — URL: `https://www.omg.org/spec/BPMN/2.0/PDF`.

11. BPMN 2.0: The state of support and implementation / M. Geiger [и др.] // Future Generation Computer Systems. — 2018. — T. 80. — C. 250—262. — DOI: `10.1016/j.future.2017.01.006`.

12. *ISO/IEC 19510:2013*. Information technology — Object Management Group Business Process Model and Notation. — Geneva, CH : International Organization for Standardization, 2013. — 498 c.

13. *Freund J.*, *Rücker B.* Real-Life BPMN: Using BPMN and DMN to analyze, improve, and automate processes in your company. — 4th. — 2019. — 229 c. — ISBN 978-1-08-630209-7.

14. *Ras J.* ETL — Extract, Transform, Load: Data Analytics Study Guide. — Church, 2018. — 602 c. — ISBN 978-1-64354-148-8.

15. *Jackson W.* JSON Quick Syntax Reference. — Apress, 2016. — 142 c. — ISBN 978-1-4842-1863-1.

16. *Baarsen J. van*. GitLab Cookbook. — Packt Publishing, 2014. — 172 c. — ISBN 978-1-78398-685-9.

17. *Sagar R.* Jira Quick Start Guide: Manage your projects efficiently using the all-new Jira. — Packt Publishing, 2019. — 162 c. — ISBN 978-1-78934-588-9.

18. *Camunda*. Camunda Best Practices. — 2021. — URL: `https://camunda.com/best-practices/_book`.

19. Camunda Platform [исходный код] / GitHub. — URL: `https://github.com/camunda/camunda-bpm-platform`.

20. *Camunda*. The Camunda Platform Manual. — Вер. 7.15.0. — 2021. — URL: `https://docs.camunda.org/manual/7.15`.

21. *Wiśniewski P.* Decomposition of business process models into reusable sub-diagrams // ITM Web of Conferences. — 2017. — T. 15. — C. 01002. — DOI: `10.1051/itmconf/20171501002`.

22. *Heckler M.* Spring Boot: Up and Running. — O'Reilly Media, 2021. — 331 c. — ISBN 978-1-4920-7695-7.

23. *Samoylov N.*, *Sanaulla M.* Java 11 Cookbook: A definitive guide to learning the key concepts of modern application development. — 2nd. — Packt Publishing, 2018. — 802 с. — ISBN 978-1-78913-528-2.

24. *Wilkinson A.*, *Frederick S.* Spring Boot Gradle Plugin Reference Guide / VMware. — Вер. 2.4.5. — 2021. —
URL: `https://docs.spring.io/spring-boot/docs/2.4.5/gradle-plugin/reference/pdf/spring-boot-gradle-plugin-reference.pdf`.

25. Spring Boot Reference Documentation / P. Webb [и др.] ; VMware. — Вер. 2.4.5. — 2021. —
URL: `https://docs.spring.io/spring-boot/docs/2.4.5/reference/pdf/spring-boot-reference.pdf`.

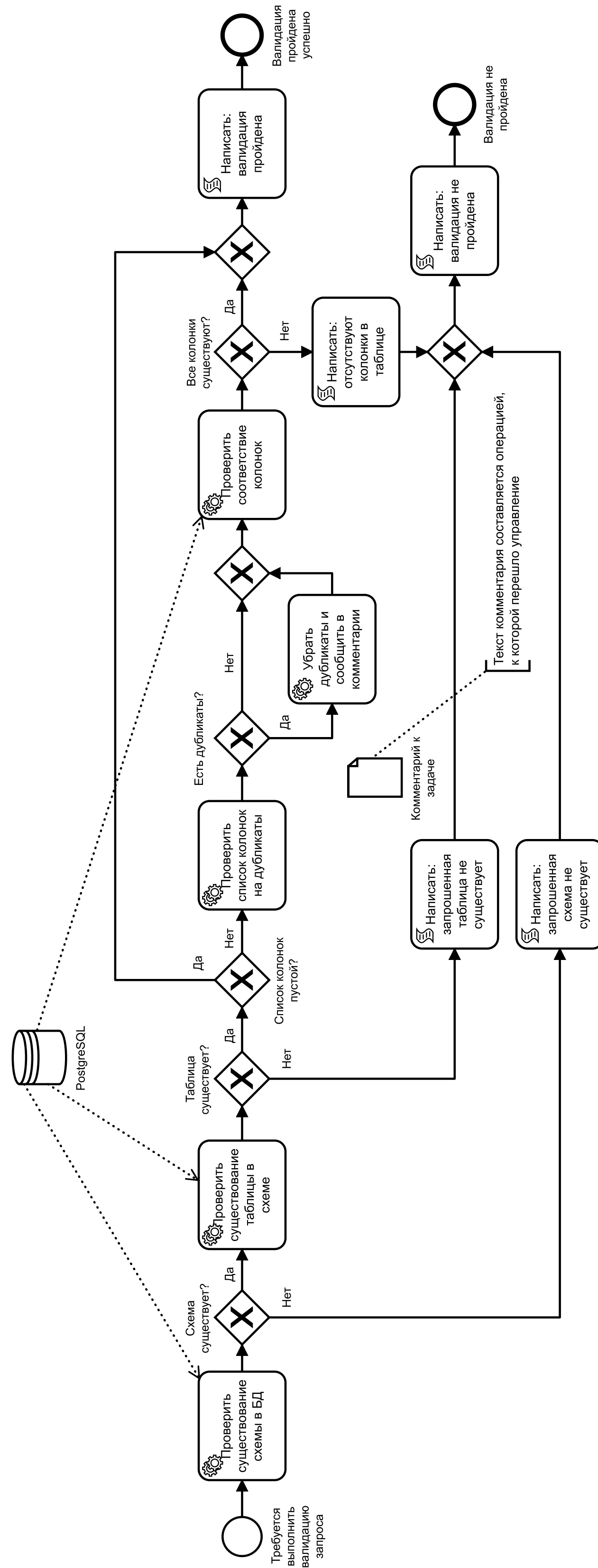
26. JIRA REST Java Client Implementation / MvnRepository. — Вер. 5.2.2. —
URL: `https://mvnrepository.com/artifact/com.atlassian.jira/jira-rest-java-client-core/5.2.2`.

27. GitLab4J API / GitHub. — Вер. 4.16.0. —
URL: `https://github.com/gitlab4j/gitlab4j-api/tree/gitlab4j-api-4.16.0`.

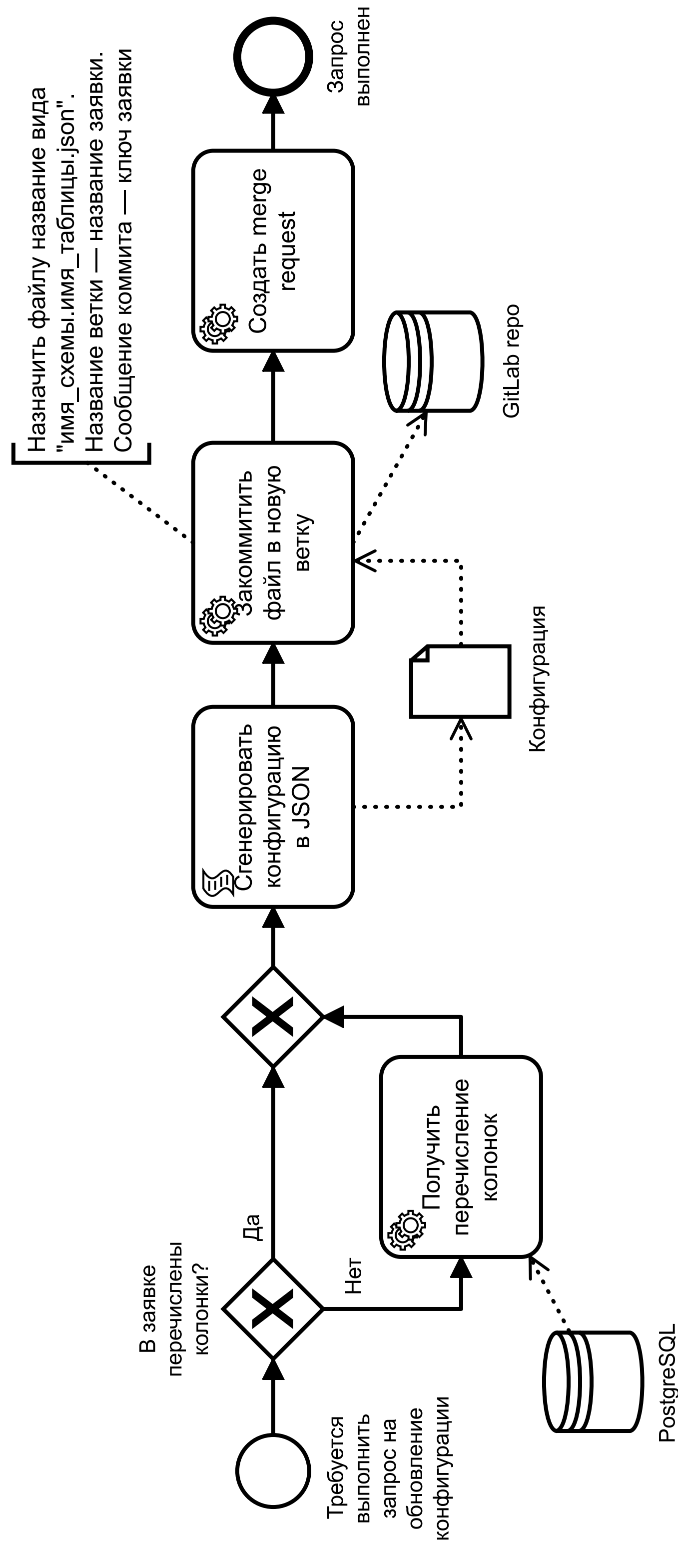
28. *Kuruvilla J.* JIRA Development Cookbook. — 3rd. — Packt Publishing, 2016. — 598 с. — ISBN 978-1-78588-633-1.

29. Configuration Update Process Automation / GitLab. —
URL: `https://gitlab.com/alexbrovkin-graduate-work-2021/configuration-update-automation`.

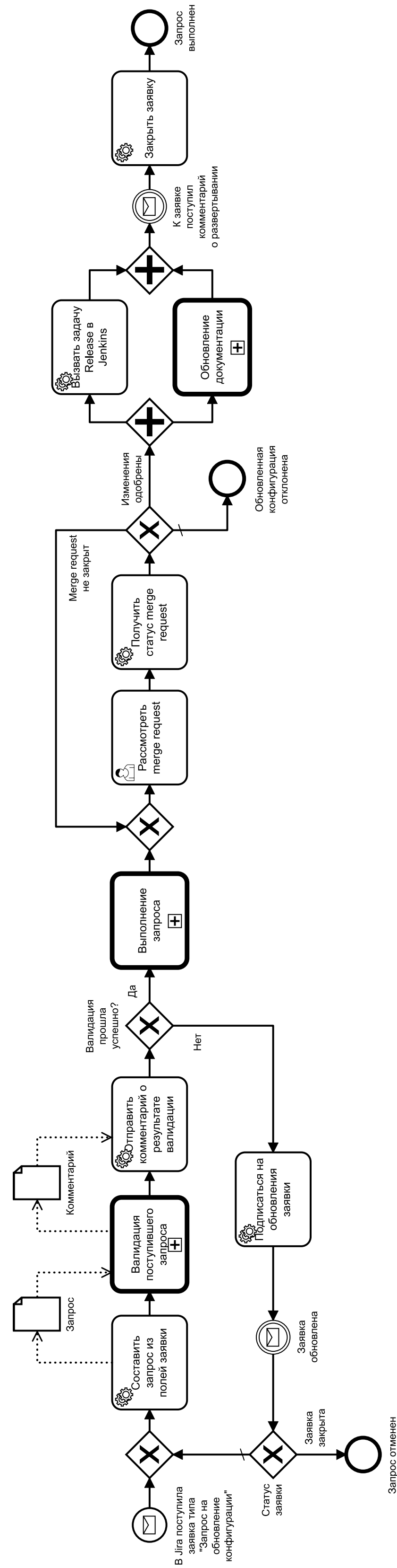
Приложение А. Модель процесса валидации запроса



Приложение Б. Модель процесса выполнения запроса



Приложение В. Модель основного процесса



Приложение Г. Исходный код интеграции с Jira

```
package automation.service;

import com.atlassian.jira.rest.client.api.IssueRestClient;
import com.atlassian.jira.rest.client.api.domain.Comment;
import com.atlassian.jira.rest.client.api.domain.Issue;
import com.atlassian.jira.rest.client.api.domain.Transition;
import com.atlassian.jira.rest.client.api.domain.input
↪ .TransitionInput;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import
↪ org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Autowired;

import java.util.List;
import java.util.stream.StreamSupport;

@Service
public class JiraIntegrationService {

  private static final Logger LOGGER =
  ↪ LoggerFactory.getLogger(JiraIntegrationService.class);

  @Autowired
  private IssueRestClient issueRestClient;

  public void addComment(String issueKey, String
  ↪ commentText) {
    Issue issue = getIssue(issueKey);
    LOGGER.info("Sending comment to issue {}", issueKey);
    issueRestClient.addComment(issue.getCommentsUri(),
    ↪ Comment.valueOf(commentText));
  }

  public Issue getIssue(String issueKey) {
    return issueRestClient.getIssue(issueKey).claim();
  }
```

```
public void setIssueToDo(String issueKey) {
  LOGGER.info("Setting \"To Do\" status for issue {}",
  ↪  issueKey);
  Issue issue = getIssue(issueKey);
  issueRestClient.transition(issue, mapTransition(issue,
  ↪  List.of("to do", "open", "reopened"))).claim();
}

public void setIssueInProgress(String issueKey) {
  LOGGER.info("Setting \"In Progress\" status for issue
  ↪  {}", issueKey);
  Issue issue = getIssue(issueKey);
  issueRestClient.transition(issue, mapTransition(issue,
  ↪  List.of("in progress"))).claim();
}

public void setIssueInReview(String issueKey) {
  LOGGER.info("Setting \"In Review\" status for issue {}",
  ↪  issueKey);
  Issue issue = getIssue(issueKey);
  issueRestClient.transition(issue, mapTransition(issue,
  ↪  List.of("in review"))).claim();
}

public void setIssueClosed(String issueKey) {
  LOGGER.info("Setting \"Done\" status for issue {}",
  ↪  issueKey);
  Issue issue = getIssue(issueKey);
  issueRestClient.transition(issue, mapTransition(issue,
  ↪  List.of("done", "closed", "resolved"))).claim();
}

private TransitionInput mapTransition(Issue issue,
↪  List<String> transitions) {
  issueRestClient.getTransitions(issue).claim()⌟
  ↪  .iterator();
  Transition transition =
```

```
      StreamSupport.stream(issueRestClient
      ↪  .getTransitions(issue).claim().spliterator(),
      ↪  false)
        .filter(
          t ->
          ↪  transitions.contains(t.getName().toLowerCase())
        ).findFirst()
        .orElseThrow(() -> new
        ↪  UnsupportedOperationException("Couldn't get
        ↪  transition"));
    return new TransitionInput(transition.getId());
  }
}
```

Приложение Д. Исходный код метода формирования запроса из заявки

```
public static ConfigurationUpdateRequest of(Issue issue) {
  LOGGER.info("Building request from issue {}",
  ↪  issue.getKey());

  Map<ConfigurationUpdateFieldType, Object> fields =
  StreamSupport.stream(issue.getFields().spliterator(),
  ↪  false)
    .filter(
      field -> fromJiraName(field.getName()) !=
      ↪  UNWANTED_FIELD && field.getValue() != null
    )
    .collect(
      Collectors.toMap(
        field -> fromJiraName(field.getName()),
        IssueField::getValue
      )
    );
  if (!fields.keySet().containsAll(REQUIRED_FIELDS)) {
    LOGGER.error("Some required fields are missing in the
    ↪  issue");
    throw new IllegalArgumentException("Some of required
    ↪  fields are not provided");
  }

  String schemaName = (String) fields.get(SCHEMA);
  String tableName = (String) fields.get(TABLE);
  int frequency = ((Double)
  ↪  fields.get(FREQUENCY)).intValue();

  boolean createSnapshots = false;
  try {
    createSnapshots =
    "yes".equalsIgnoreCase(
    ((JSONObject)
    ↪  fields.get(CREATE_SNAPSHOTS)).getString("value")
    );
```

```
} catch (JSONException e) {
  LOGGER.warn("Something wrong with \"Extraction type\"
  ↪  field. Check your Jira Custom Fields settings." +
  " Default value (false) has been assigned", e);
}

Collection<Column> columns = new ArrayList<>();
String columnsFieldStr = (String) fields.get(COLUMNS);
try {
  if (columnsFieldStr != null) {
    if (!COLUMNS_FIELD_CORRECT_PATTERN
    ↪  .matcher(columnsFieldStr).matches())
    ↪  {
      throw new IllegalArgumentException("\"Set of
      ↪  columns\" field doesn't match ");
    }
    Matcher columnsItemsMatcher =
    ↪  COLUMNS_ITEMS_PATTERN.matcher(columnsFieldStr);
    while (columnsItemsMatcher.find()) {
      String name = columnsItemsMatcher.group("name");
      String type = columnsItemsMatcher.group("type")
      ↪  .replaceAll("\\s{2,}", "
      ↪  ");
      String length = columnsItemsMatcher.group("length");
      Column currentColumn;
      if (length != null) {
        currentColumn = new Column(name, type,
        ↪  Integer.valueOf(length));
      } else {
        currentColumn = new Column(name, type);
      }
      columns.add(currentColumn);
    }
    LOGGER.info("\"Set of columns\" field parsed
    ↪  successfully");
  } else {
    LOGGER.info("No columns to parse");
  }
} catch (IllegalArgumentException e) {
```

```
    LOGGER.warn("Couldn't parse \"Set of columns\" field due
    ↪  to incorrect format", e);
  }

  return new ConfigurationUpdateRequest(issue.getSummary(),
    Objects.requireNonNull(issue.getReporter())
      .getDisplayName(),
    schemaName, tableName, columns, frequency,
    ↪  createSnapshots);
}
```

Приложение Е. Исходный код интеграции с GitLab

```
package automation.service;

import automation.domain.GitLabProperties;
import org.gitlab4j.api.GitLabApi;
import org.gitlab4j.api.GitLabApiException;
import org.gitlab4j.api.models.CommitAction;
import org.gitlab4j.api.models.CommitPayload;
import org.gitlab4j.api.models.MergeRequest;
import org.gitlab4j.api.models.MergeRequestParams;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import
↪  org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Autowired;

@Service
public class GitLabIntegrationService {

  private static final Logger LOGGER = LoggerFactory⌋
  ↪  .getLogger(GitLabIntegrationService.class);

  @Autowired
  private GitLabApi gitLabApi;
  @Autowired
  private GitLabProperties gitLabProperties;

  public void commitFile(String branchName, String
  ↪  commitMessage, String fileName, String fileContent)
  ↪  throws GitLabApiException {
    LOGGER.info("Committing file '{}' to branch '{}'",
    ↪  fileName, branchName);
    boolean fileExists;
    try {
      gitLabApi.getRepositoryFileApi()⌋
      ↪  .getFile(gitLabProperties.getRepoPath(), fileName,
      ↪  gitLabProperties.getStartBranch());
      fileExists = true;
```

```
      LOGGER.info("File '{}' already exists and will be
      ↪  updated", fileName);
    } catch (GitLabApiException e) {
      if (e.getHttpStatus() == 404) {
        LOGGER.info("File '{}' doesn't exist and will be
        ↪  created", fileName);
        fileExists = false;
      } else {
        throw e;
      }
    }
    CommitAction commitAction = new CommitAction()
      .withFilePath(fileName)
      .withContent(fileContent)
      .withAction(fileExists ? CommitAction.Action.UPDATE :
      ↪  CommitAction.Action.CREATE);

    gitLabApi.getCommitsApi().createCommit(
      gitLabProperties.getRepoPath(),
      new CommitPayload()
        .withBranch(branchName)
        .withStartBranch(gitLabProperties.getStartBranch())
        .withCommitMessage(commitMessage)
        .withAction(commitAction)
    );
  }

  public MergeRequest createMergeRequest(String
  ↪  sourceBranch) throws GitLabApiException {
    LOGGER.info("Creating merge request for branch '{}'",
    ↪  sourceBranch);
    String title = sourceBranch.toUpperCase();
    return
    ↪  gitLabApi.getMergeRequestApi().createMergeRequest(
      gitLabProperties.getRepoPath(),
      new MergeRequestParams()
        .withSourceBranch(sourceBranch)
        .withTargetBranch(gitLabProperties.getStartBranch())
        .withTitle(title)
```

```
        .withRemoveSourceBranch(true)
    );
  }

  public boolean checkIfMerged(int mergeRequestIid) throws
  ↪  GitLabApiException {
    return "merged".equalsIgnoreCase(
      gitLabApi.getMergeRequestApi()⌋
      ↪  .getMergeRequest(gitLabProperties.getRepoPath(),
      ↪  mergeRequestIid).getState()
    );
  }

  public boolean checkIfOpen(int mergeRequestIid) throws
  ↪  GitLabApiException {
    return "opened".equalsIgnoreCase(
      gitLabApi.getMergeRequestApi()⌋
      ↪  .getMergeRequest(gitLabProperties.getRepoPath(),
      ↪  mergeRequestIid).getState()
    );
  }

  public static String convertStringToBranchName(String s) {
    //https://wincent.com/wiki/Legal_Git_branch_names
    return s.replaceAll("\\s", "_")
      .replaceAll("^\\.|\\.\\⌋
      ↪  .|\\\\|[\\000-\\037\\0177*~^:?\\[]|.lock$|@\\{",
      ↪  "")
      .toLowerCase();
  }
}
```