Министерство образования и науки
Российской Федерации

Московский авиационный институт
(национальный исследовательский университет)

Институт компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

ЖУРНАЛ

ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

Наименование практики: *вычислительная/исследовательская*
Студент: А. А. Савин
Факультет №8, курс 3, группа 7

Практика с 29.06.22 по 12.07.22

Москва, 2022

ИНСТРУКЦИЯ

о заполнении журнала по производственной практике

Журнал по производственной практике студентов имеет единую форму для всех видов практик.

Задание в журнал вписывается руководителем практики от института в первые три-пять дней пребывания студентов на практике в соответствии с тематикой, утверждённой на кафедре до начала практики. Журнал по производственной практике является основным документом для текущего и итогового контроля выполнения заданий, требований инструкции и программы практики.

Табель прохождения практики, задание, а также технический отчёт выполняются каждым студентом самостоятельно.

Журнал заполняется студентом непрерывно в процессе прохождения всей практики и регулярно представляется для просмотра руководителям практики. Все их замечания подлежат немедленному выполнению.

В разделе «Табель прохождения практики» ежедневно должно быть указано, на каких рабочих местах и в качестве кого работал студент. Эти записи проверяются и заверяются цеховыми руководителями практики, в том числе мастерами и бригадирами. График прохождения практики заполняется в соответствии с графиком распределения студентов по рабочим местам практики, утверждённым руководителем предприятия. В разделе «Рационализаторские предложения» должно быть приведено содержание поданных в цехе рационализаторских предложений со всеми необходимыми расчётами и эскизами. Рационализаторские предложения подаются индивидуально и коллективно.

Выполнение студентом задания по общественно-политической практике заносятся в раздел «Общественно-политическая практика». Выполнение работы по оказанию практической помощи предприятию (участие в выполнении спецзаданий, работа сверхурочно и т.п.) заносятся в раздел журнала «Работа в помощь предприятию» с последующим письменным подтверждением записанной работы соответствующими цеховыми руководителями. Раздел «Технический отчёт по практике» должен быть заполнен особо тщательно. Записи необходимо делать чернилами в сжатой, но вместе с тем чёткой и ясной форме и технически грамотно. Студент обязан ежедневно подробно излагать содержание работы, выполняемой за каждый день. Содержание этого раздела должно отвечать тем конкретным требованиям, которые предъявляются к техническому отчёту заданием и программой практики. Технический отчёт должен показать умение студента критически оценивать работу данного производственного участка и отразить, в какой степени студент способен применить теоретические знания для решения конкретных производственных задач.

Иллюстративный и другие материалы, использованные студентом в других разделах журнала, в техническом отчёте не должны повторяться, следует ограничиваться лишь ссылкой на него. Участие студентов в производственно-технической конференции, выступление с докладами, рационализаторские предложения и т.п. должны заноситься на свободные страницы журнала.

Примечание. Синьки, кальки и другие дополнения к журналу могут быть сделаны только с разрешения администрации предприятия и должны подшиваться в конце журнала.

Руководители практики от института обязаны следить за тем, чтобы каждый цеховой руководитель практики перед уходом студентов из данного цеха в другой цех вписывал в журнал студента отзывы об их работе в цехе.

Текущий контроль работы студентов осуществляется руководители практики от института и цеховыми руководителями практики заводов. Все замечания студентам руководители делают в письменном виде на страницах журнала, ставя при этом свою подпись и дату проверки.

Результаты защиты технического отчёта заносятся в протокол и одновременно заносятся в ведомость и зачётную книжку студента.

Примечание. Нумерация чистых страниц журнала проставляется каждым студентом в своём журнале до начала практики.

С инструкцией о заполнении журнала ознакомились:

« » _____ 2022 г.
(дата)

Студент Савин А. А. _____
(подпись)

ЗАДАНИЕ

Создать приложение для просмотра информации о книгах с возможностью добавлять в каталог книги с помощью API. Все это необходимо сделать, используя Java фреймворк Spring Boot и библиотеку Hibernate. Для создание контейнеров использовать Docker.

Руководитель практики от института:

« » _____ 2022 г. Кухтичев А. А. _____

(дата) (подпись)

ТАБЕЛЬ ПРОХОЖДЕНИЯ ПРАКТИКИ

Дата	**Содержание или наименование проделанной работы**	**Место работы**	**Время работы**		**Подпись цехового руководителя**
			Начало	Конец	
29.06.2022	Получение задания	МАИ	9:00	18:00	
01.07.2022	Изучение документации по Spring Boot и Hibernate	МАИ	9:00	18:00	
02.07.2022	Написание простого сайта-макета	МАИ	9:00	18:00	
03.07.2022	Создание отображающей компоненты	МАИ	9:00	18:00	
04.07.2022	Начало работ с API	МАИ	9:00	18:00	
05.07.2022	Доработка компоненты-загрузчика	МАИ	9:00	18:00	
06.07.2022	Создание JSON парсера из файла	МАИ	9:00	18:00	
07.07.2022	Создание JSON парсера с url с использованием API ключа	МАИ	9:00	18:00	
09.07.2022	Чтение документации по Docker и мелкие исправления имеющейся программы	МАИ	9:00	18:00	
10.07.2022	Работа с Docker	МАИ	9:00	18:00	
11.07.2022	Создание отчета и презентации	МАИ	9:00	18:00	
12.07.2022	Сдача журнала	МАИ	9:00	18:00	

Отзывы цеховых руководителей практики

Студент Савин А. А. разработал ...

Презентация защищена на комиссии кафедры 806. Работа выполнена в полном объёме. Рекомендую на оценку « ». Все материалы сданы на кафедру.

ПРОТОКОЛ

ЗАЩИТЫ ТЕХНИЧЕСКОГО ОТЧЁТА

по *производственной практике*

студентами: Савин Александр Андреевич

Слушали:	**Постановили:**
Отчёт практиканта	считать практику выполненной и защищённой на Общая оценка: _____

Руководители: Зайцев В. Е. _____

Кухтичев А. А. _____

Дата: 12 июля 2022 г.

МАТЕРИАЛЫ ПО РАЦИОНАЛИЗАТОРСКИМ ПРЕДЛОЖЕНИЯМ

ТЕХНИЧЕСКИЙ ОТЧЁТ ПО ПРАКТИКЕ

Архитектура

Программа состоит из нескольких компонент. Помимо общего разделение на две отдельных компоненты по функционалу существует и отдельная база данных, к которой происходит подключение.
База данных *MySQL* с названием *books_db* хранится на локальном сервере и содержит всю необходимую информацию о книгах, которые обрабатываются двумя компонентами. Информация о книге представлена несколькими полями типа *Integer* или *Varchar*(255). Таблица имеет следующие поля: *id* - номер книги в таблице (поле имеет ценность лишь внутри таблицы), *title* - название книги на языке оригинала, это поле первое, что выводится в краткой сводке о книге не главной странице, где отображается список книг. *Publisher* - информация об издателе, представленная на сервисе Google Books, данное поле часто бывает пустым, так как информации сервиса не всегда достаточно. *publishedDate* - данные о дате публикации, хранится в строковом формате, так как бывают различные форматы даты. *description* - краткое описание книги, здесь приходится обрезать текст до 255 символов, так как MySQL таблица не позволяет хранить строки бОльшей длины. *language* - поле, хранящее сокращение, обозначающее язык, на котором написана книга, обычно две буквы. *googleId* - *ID*, присвоенный данной книге сервисом Google Books, служащий своеобразным аналогом ISBN, который не всегда возможно использовать ввиду отсутствия единого реестра для всех книг. Переменные *categories* - категории по жанрам, к которым относится книга, *authors* - список авторов книги с их инициалами, изначально подаются в качестве списка, но с помощью метода *toString* приводятся к строке и уже в таком виде хранятся в таблице, так как типа данных *MySQL* не предлагает.
Программа состоит из двух компонент: *JsonBookParser* и *Library*.

Первая компонента занимается чтением данных о книгах из файлов формата *.json* и с сайта *GoogleBooks* посредством api ключа. JsonBookParser состоит из класса *Book* в котором размещены все поля для объекта книги и прописаны необходимые для создания таблицы аннотации. Аннотации позволяют создать таблицу с нужными полями уже при первой компиляции. С помощью аннотаций был настроен инкремент для поля *id* внутри базы данных. Для удобства и лаконичности кода я также использовал библиотеку *Lombok*, которая позволяет посредством двух аннотаций сгенерировать необходимые *gettersandsetters* для всех полей. Программа имеет довольно типичную для *SpringBoot* приложений архитектуру. Для общения с пользователем необходим контроллер. Внутри контроллера прописывается обработка запросов к программе посредством аннотации *GetMapping*. Функционал у парсера разнообразный, потому нужно было сделать два контроллера: *FileParserController* и *URLParserController*. *FileParserController* отвечает за обработку запроса по парсингу из файла. Он принимает *Mapping*, извлекает параметр, представляющий собой адрес файла, и передает его дальше в программу. Внутри функции, параметр передается в созданный объект класса *JsonBookParser*.
Класс *JsonBookParser* занимается обработкой всех действий, связанных с парсингом *Json*-файлов. Он содержит постоянную переменную *APIKey*, которая недоступна извне и является уникальной для этой программы, так как с помощью нее есть возможность получать информацию с сервиса *GoogleBooks*. Помимо этого поля в классе присутствуют приватные поля, отвечающие за *URL* адрес обращения. Они изменяемы через конструктор, но, по-хорошему, изменять есть необходимость только параметр *website_q*, который нужен для корректного задания запроса к сервису. Все операции обрабатываются исключениями. Объект типа *URL* создается в функции *URLBuilder* и нужен для корректной обработки запросов. Для работы с *.json*-файлами используются метод *readJsonIntoString*, который информацию, полученную по *API* в строку с дальнейшей возможностью записать эту строку в файл или начать ее расшифровывать. Эта функция используется в других, например в *readJsonFromURLToList*, где происходит чтение *JSON* в строку, откуда происходит извлечение данных по ключу *items*. Все ключи получены опытным путем, во время просмотра

содержимого $Json$-файла. Найти способ автоматизировать этот процесс не удалось. Извлечение $List$ из файла происходит в функции $readJsonFromFileToList$. Названия делались говорящими, чтобы проще было использовать парсер. Основная работа по извлечению информации происходит в $extractBookListFromJsonArray$. Здесь происходит обращение по ключам ко всем зашифрованным данным, с соответствующей проверкой на пустоту. После эти данные преобразуются в объект класса $Book$ и добавляются в $List$, который впоследствие передается обратно из функции. Для полноты класса существует и функция $writeJsonToFile$, которая записывает информацию с сайта в файл с расширением $.json$.
Так как программа имеет два возможных способа чтения информации, были созданы два контроллера: $FileParserController$ и $URLParserController$. Эти контроллеры отвечают за обработку запроса в строке поиска браузера. Посредством аннтоации $GetMapping$ и $@equestParam$ передаются параметры $path$ для $FileParserController$ и q для $URLParserController$.
Каждый контроллер имеет объект класса $BookParserService$. Объекты этого класса отвечают за бизнес-логику приложения, здесь вносятся коррективы в стандартные функции по чтению и записи в БД, так, например, здесь присутствует проверка на существования объекта с данным ID, чтобы избежать записи идентичных объектов. Класса $BookParserService$ в свою очередь имеет объект интерфейса $BookParserRepository$, который наследует $JpaRepository < Book, Long >$. Этот класс необходим для работы с самой базой, чтобы не прописывать методы общения с ней вручную.

Вторая компонента программы занимается чтением данных из БД и отображением их для пользователя. В ней, так же как и в первой компоненте, содержится класс Book, снабженный необходимыми аннотациями для работы с БД и генерацией $gettersandsetters$. $BookRepository$, интерфейс, наследующий $JpaRepository < Book, Long >$, где $Book$ - тип хранящегося объекта, а $Long$ - тип $primary - key$. $BookService$ - такой же как и в первой компоненте класс-сервис.
Контроллеров в этой программе четыре. $MainPageController$ отвечает за работу с главной страницей. Он принимает пустой запрос к $localhost$ и отправляет пользователя на специальную $html$ страницу $main - page$. К этой $html$ странице прикреплены шаблоны $header$ и $footer$, которые одинаковы для всех страниц и поэтому вынесены в отдельный пакет. Также этот контроллер передает строковый параметр, являющейся обычный приветствием.
Второй контроллер $BookListController$ отвечает за работу с списком книг. Он обрабатывает запрос $/book - list$ и отображает соответствующую страницу. В функции-обработчике создается объект типа $Iterable < Book >$. Этот объект принимает результат работы функции $findAll$ объекта типа $BookService$. Этот результат передается в $html$ страницу $book - list$. Там из этого списка через цикл $foreach$ извлекаются данные и читаются поля элементов списка. Впоследствие значения полей выводятся на экран. Это и есть представление данных для пользователя. Также на этой $html$ странице в каждой ячейке каталога создается кнопка для вызова функции, отвечающей за просмотр каждой книги отдельно.
Третий контроллер $InfoFormController$ обрабатывает запрос на просмотр книги из БД по ее внутреннему ID. Контроллер обрабатывает соответствующий $Mapping$: $/book - list/id$, где id - параметр для базы данных, чтобы по нему получить всю нужную информацию о книге. Так предварительно проверив существование объекта с данным ID с помощью функции $existsById(id)$ можно вызвать функцию $bookService.findById(id)$, возвращающей объект типа $Optional < Book >$, его можно передать в $html$ страницу $info - form$. На этой странице происходит вывод всех полей объекта. При отсутствии объекта с заданным id в базе будет вызван $redirect$, отправляющий пользователя обратно на главную страницу.
Четвертый контроллер скрыт от обычного пользователя и был прописан изначально с целью проверки работы с базой. Контроллер называется $AddFormController$. Он вызывается при мэппинге $/add - form$ и обрабатывает получение параметров от $html$ страницы. В этом контроллере два метода, первый по мэппингу $/add - form$ вызывает $html$ страницу $add - form$, а второй, под названием $addBook$ принимает параметры с аннотациями $RequestParam$. Эти аннотации помогают обработать

данные, получаемые в контроллер. Здесь же вызывается конструктор для создания книги и сохранения этой книги через *BookService*.

Стоит отметить, что все классы: сервисы, контроллеры, репозитории имеют аннотации, соответствующие их функционалу, например: *Controller*, *Service*, *Repository*. Помимо этого, все конструкторы снабжены аннотациями *Autowired*, которые помогают *SpringBoot* самостоятельно организовывать все межклассовое взаимодействие.
Мне удалось использовать *Docker* лишь для компоненты $JsonBookParser$, для организации работы сервера. Для работы нужно было создать $dockerfile$, в котором прописывалась версия $MySQL$ и пароль для пользователя *root*. Также был создан файл $docker - compose$, где были указаны все параметры создаваемого контейнера, вроде номера порта, который пришлось поменять и названия самого выходного контейнера. $docker - compose$ имеет расширения *.yaml*, так как оно позволяет сделать код более наглядным и местами сократить написание одного и того же обращения к функции с помощью деления кода на блоки. После этого пришлось скомпилировать получившийся файл и получить его внутри программы *DockerDekstop*. Для компиляции самого проекта использовался *Maven* с настроенными параметрами имени файла после сборки и добавленными зависимостями для использования различных библиотек, упрощающих написание кода, а также необходимых для подключения к БД *MySQL* и других разнообразных манипуляций. Для начальной сборки приложения использовал специальный *springinitializr*, позволяющий удобно и быстро настроить версии языка *Java*, имена пакетов и сразу подключить необходимые зависимости для *Maven* и *Gradle*.

Описание

Данная программа представляет собой простое библиотечной приложение нужное для просмотра книг из некоего каталога и пополнения этого каталога с помощью запросов к *GoogleBooks* и собственных *Json* файлов. Изначально пользователь попадает на простой сайт с шапкой, где наверху располагаются две кнопки: *MainMenu* и *BookList*. Кнопка, отвечающая за главное меню, ведет пользователя на почти пустую страницу, где выводится простое приветствие. *BookList* выводит пользователя на страницу, где отображается список всех книг в базе по порядку их добавления. Книги представляются как небольшой прямоугольник, где написаны название книги, авторы и уникальный *GoogleId* полученный от сервиса. Естественно, этой информации мало для полноценного каталога, поэтому пользователю предлагается пройти по ссылке, хранящейся в этом прямоугольнике информации и рассмотреть книгу подробнее. На странице по запросу $/book-list/id$, где id - внутренний номер книги, можно с помощью простого интерфейса увидеть все известные о книге данные. К сожалению, часть полей будет не заполнено и обозначено как $NoData$, так как с помощью $JsonBookParser$ можно получить доступ по API только к сервису *GoogleBooks*, который не всегда хранит абсолютно всю информацию о книгах. Такое представление каталога минималистично отображает всю необходимую информацию и имеет весь необходимый функционал для просмотра. Компонента, отвечающая за добавление книг, выглядит сложнее, так как не предназначена для обычного пользователя. Здесь предполагается, что некий модератор будет вбивать запросы в поисковую строку браузера и добавлять книги в базу. Попытки добавить уже существующие книги не будут пресекаться, но не будут проходить внутри серверной части приложения. Для компоненты по добавлению книг есть два вида запросов. В первом, предполагается, что модератор будет задавать аргумент с говорящим названием $filepath$. В этот аргумент передается адрес *.json* файла внутри операционной системы, к этому файлу программа будет обращаться и извлекать из него данные для загрузки их в БД. Также есть возможность добавлять данные напрямую из сервиса *GoogleBooks*. Здесь для сервиса существует конкретный запрос: $https://www.googleapis.com/books/v1/volumes?q=X:keyes\&key=Y$", где на месте X должен стоять некоторый запрос вроде имени автора или названия темы, а вместо Y должен быть уникальный API-ключ. Подробнее о запросах можно посмотреть на сайте *Google*, посвященному работе с API, но для модератора достаточно лишь воводить название книги или автора, чтобы

добавить все имеющиеся книги по данному запросу в каталог. Уникальный API ключ не может быть изменен и вшит внутрь программы без опций его каким-либо способом изменить. Интерфейс этой компоненты практически полностью отсуствтвует и выдает лишь название операции по добавлению информации в каталог при успешном завершении операции добавления данных.

Реализация

Основным языком для разработки данного приложения является Java. Этот язык мною был выбран ввиду его частого использования для бэкенд разработки и личных предпочтений. Фреймворком для сборки был выбран Maven, так как лично мне он кажется более удобным и наглядным чем Gradle. Код подключения необходимых зависимостей для него также легко найти. Для Java существует ряд фреймворков, одним из которых является Spring Boot, выбранный мною для разработки данного приложения. Для данного фреймворка существует обширная и удобная документация, к тому же про него много туториалов на *YouTube*. Для подключения к базе данных у Java существует библиотека Hibernate, сильно упрощает взаимодействие с БД и дает возможность не использовать SQL для написания команд. К тому же она также поддерживает функционал аннотаций, который использует и Spring Boot. Для создания сервера с базой данных был использован клиент программы MySQL, который позволяет наглядно общаться с БД посредством SQL-запросов или вручную выставлять конфигурацию вроде настройки типов полей таблицы. Подобный функционал не уникальный, но этот клиент мне попался на глаза первым, хотя для работы можно было использовать другие программы управления БД вроде MongoDB. В качестве среды разработки я выбрал Intellij Idea, которая стабильнее и удобнее чем конкуренты, к тому же с продуктами этой компании я имел дело ранее. Также данная среда разработки поддерживало быстрое у удобное соединение с БД на MySQL. Для создание контейнера с сервером я использовал программу Docker. Для работы с ней были прописаны конфигурационные файлы внутри программы по парсингу книг. Docker позволяет удобно передать базу данных со всеми ее настройками другим, поэтому опыт работы с ней полезен. Для конфигурации Dockerfile я использовал расширение *.yaml*, так как оно позволяет не прописывать по нескольку раз один и тот же код.

Тестирование

Программа тестировалась мною на различных аргументах. Сначала тестировалась часть, необходимая для парсинга. Для этого я создал несколько *.txt* файлов (для использования блокнота в качестве редактора) и записал в них большое количество данных о книгах, скопированных мною с страниц, выдаваемых по соответствующим запросам в *google*. После записи я изменил расширение *.txt* на *.json* и начал тестировать программу, вручную вбивая адреса файлов. Затем я перешел к тестированию части, считывающей информацию с *Googlebooks*. Для этого я задавал параметр в строку запроса и выполнял программу, после чего самостоятельно переходил по сгенерированной в классе *JsonBookParser* ссылке и сверял выданные данные с теми, что были записаны в таблицу. Естественно сверять дословно такие объемы информации трудно, поэтому интерес представляло количество выдаваемых книг.
Для тестирования читающей компоненты, я использовал клиент *MySQL*, где посредством команды $SELECT * FROM books$ можно узнать содержимое таблицы и сверить его с тем, что выдает в окно браузера моя программа. Вообще в тестировании клиент *MySQL* сильно помогал, так как понятно отображал информацию из таблицы, хотя измененная очередность полей иногда раздражала.

Ссылка на GitHub

https://github.com/Brinckley/JavaSummerPractice/tree/main/library https://github.com/Brinckley/JavaSum

Вывод

Выполняя данную практику, я получил опыт во многих сферах. Для начала усовершенствовал свои навыки в *Java*, предварительно подтянув *JavaCore*, хотя почти все не понадобилось, так как основная часть проекта выполнялась за счет специального функционала, а бизнес-логика была достаточно простая, чтобы использовать многопоточность или лямбда-выражения. У меня был опыт создания приложения с фреймворком *Spring*, но с усовершенствованной версией работать не приходилось, поэтому я рад, что получил такой опыт сейчас. *SpringBoot* оказался интересным способом упростить себе жизнь, работая с *web*-приложениями вроде данной программы-книжного каталога. Библиотека *Hibernate* также вызывала мой интерес, поэтому хорошо, что мне пришлось с ней работать, ведь это очень полезная для взаимодействия с БД вещь, направленная на упрощение всех манипуляций с данными. Работа с серверами *MySQL* меня тоже заинтересовала, хотя в куда меньшей степени, так как это было монотонно и долго пришлось настраивать программу из-за нескольких мелких проблем. Помимо всего, я получил опыт в работе с *Docker*, который является широко-востребованной программой. Разбираться в его функционале было интересно, хотя местами было трудно.