Санкт-Петербургский государственный университет
Физический факультет
Кафедра вычислительной физики

Разработка структуры Java-версии программного комплекса distolymp

Бакалаврская работа студента
дневного отделения
Тимашова Даниила Александровича

Научный руководитель:
к. ф.-м. н., доцент М**онахов В. В.**

Рецензент:
д. ф.-м. н., проф. **Валиев Ф. Ф.**

Санкт-Петербург
2017

Оглавление

Введение

В настоящее время организаторы интернет-олимпиады по физике (СПбГУ, Университет ИТМО и Южный Федеральный Университет) для ее проведения используют систему distolymp [1]. Серверную часть данной системы можно классифицировать как LAMP (Linux, Apache, MySQL, PHP). Это довольно популярная разновидность структуры комплекса серверного ПО. Такой выбор во время создания системы был продиктован не только надежностью этих средств, но и тем, что они свободно распространяются и гибки в использовании [2].

Однако, развитие системы привело к ее расширению и усложнению. Как известно, крупные веб-проекты становятся крайне запутанными при написании их на PHP. Поэтому было решено создать аналог серверной части на Java. Это обусловлено не только удобством при дальнейшем расширении distolymp, но и пользой для студентов, так как статистика показывает высокую востребованность данного языка среди работодателей. Кроме того, для студентов кафедры вычислительной физики читается курс именно по Java, и дальнейшая работа с проектом на нем гораздо удобнее и логичнее, чем на PHP.

В связи с этим, *целью данной работы* являлась разработка структуры Java-версии программного комплекса distolymp и создание приложения, показывающего работоспособность данной структуры.

Для достижения поставленной цели решались следующие *задачи*:

- анализ технологий, необходимых для написания данного приложения;
- создание приложения и проверка его работы в ОС Windows;
- установка необходимого ПО на production-сервер;
- проверка работы приложения на production-сервере.

1. Программный комплекс distolymp и причины разработки его Java-версии

1.1 Программный комплекс distolymp и виртуальные лаборатории по физике

Для реализации идеи проведения олимпиад школьников по физике в свое время был разработан программный комплекс distolymp. Он должен был решать проблемы, возникающие в процессе проведения олимпиады. Так как Интернет-олимпиада подразумевает использование интернета, то неудивительно, что distolymp реализует клиент-серверную модель взаимодействия. В нашем случае клиентом являются участники олимпиады (школьники и учителя) или студенты (в случае использования distolymp для занятий в университете). Серверная часть же состоит из двух серверов с необходимым программным обеспечением, что позволяет не бояться организаторам олимпиады ситуаций, когда один из серверов может выйти из строя.

В процессе разработки системы решались следующие проблемы:

- упрощение действий, связанных с установкой клиентской части клиентом;
- платформонезависимость клиентской части;
- независимость структуры модели от ее параметров, причем эти параметры можно менять для каждого случая;
- надежность и безопасность серверной части.

Все это привело к той структуре distolymp, которая существует в настоящее время. С точки зрения программного обеспечения distolymp состоит из ОС Debian (разных версий на двух серверах), веб-сервера Apache, СУБД MySQL, PHP 5 [3]. Такой выбор программ был вызван их популярностью, надежностью и свободным распространением. Для удобства установки данных программных средств в 2014 году М. Зуганом был создан инсталлятор [4] .

1.2 Структура программного комплекса distolymp

PHP-подсистема

Язык PHP (PHP: Hypertext Preprocessor) является языком, предназначенным для разработки приложений на стороне сервера. Помимо использования в виде файлов, написанных исключительно на PHP, его можно встраивать в HTML-документ для того, чтобы сервер перед отправкой клиенту страницы мог произвести необходимые изменения с ней. Преимущество языка в его традиционности (многие конструкции взяты из языков C и Perl), простоте и эффективности (быстрая обработка сценариев, благодаря транслирующему

интерпретатору).

Данная подсистема решает задачу связи между клиентом и базой данных. Кроме того, она способна выполнять другие действия, связанные со взаимодействием с почтовыми клиентами. Для удобства дальнейшего развития и текущего обслуживания, проект PHP-подсистемы состоит из слабосвязанных частей, каждая из которых решает свою задачу. Например, подсистема назначения дипломов отвечает за создание диплома для данного победителя или призера олимпиады [5].

База Данных

Основная задача базы данных — это хранение и предоставление удобного и быстрого доступа к данным различного типа. Обычно для взаимодействия с базами данных используют СУБД (Система управления базами данных). В нашем случае, была выбрана MySQL — это наиболее популярная СУБД в веб-разработке. Кроме того, она достаточно надежна и быстра.

Таким образом, общая схема работы комплекса такова: клиент посылает запрос, который принимает web-сервер, web-сервер передает его PHP-подсистеме, после обработки запроса от клиента подсистема формирует запрос для сервера MySQL на специально предназначенном для этого языке SQL. Результаты, полученные от базы данных, снова обрабатываются PHP-подсистемой, которая формирует HTML-страницу и передает ее web-серверу, который отсылает эту страницу клиенту.

В данный момент база данных distolymp состоит из 48 таблиц и 4 представлений. В таблицах содержится такая информация, как: данные пользователей, списки заданий, ответы пользователей и т. д.

Клиентская подсистема — BarsicPlayer, aBarsicPlayer и модели виртуальных лабораторий

Очевидно, что клиентская подсистема должна предоставлять участникам олимпиады доступ к различного рода задачам. И если в случае тестов и теоретических задач можно обойтись стандартными инструментами front-end разработки, то для экспериментальных задач сделать это гораздо сложнее. Поэтому используется специальная программа — BarsicPlayer или aBarsicPlayer (для Android-устройств) для того, чтобы «проигрывать» экспериментальные задания. При выполнении таких заданий BarsicPlayer скачивает специальный brc файл, причем параметры задания автоматически генерируются на сервере. Затем, после выполнения, пользователь отправляет данные на сервер, для этого тоже используется BarsicPlayer, который составляет Http-запрос. На сервере происходит проверка, во время которой используется описанная ранее база данных. Результаты выполнения отсылаются обратно.

Очень важно, что BarsicPlayer берет на себя часть работы при проведении олимпиады, что снижает нагрузку на сервер. Помимо возможности работы с моделями, BarsicPlayer позволяет также производить расчеты и строить графики, широко используя возможности языка программирования BARSIC, на котором он и написан.

1.3 Основные особенности серверных технологий Java

Как известно, существовало 4 программные платформы Java:

- Java Card: необходима для обслуживания смарт-карт и других устройств с малым объемом памяти;
- Java ME (Micro Edition): работа с мобильными устройствами, у которых небольшой дисплей и батарея, а также ограниченный объем памяти, к примеру КПК;
- Java SE (Standard Edition): основная технология для приложений под персональные компьютеры;
- Java EE (Enteprise Edition): создание клиент-серверных бизнес-приложений, фактически Java SE + специализированное API (Application Programming Interface).

За последние годы компания Oracle, занимающаяся поддержкой технологий Java, взяла курс на развитие таких технологий, как Embedded (поддержка программного обеспечения встроенных устройств) и Cloud (разработка программ, расположенных в облаке). Фактически, технологии Java ME и Java Card получили свое логическое продолжение в Embedded, куда, впрочем, частично вошли и другие технологии Java. При этом Java EE и Java SE продолжают существовать и как отдельные платформы.

Нас, тем не менее, интересует в первую очередь Java EE, которая применяется для решения задач разработки крупных серверных приложений. Рассмотрим основные понятия, которые используются в Java EE[6]. Кирпичиками любого крупного приложения являются *компоненты*, к примеру это могут быть апплет или небольшое веб-приложение. Фактически, компонент может работать и самостоятельно, но решать при этом лишь небольшой список задач. Для того, чтобы получить доступ к *службам*, которые сильно расширят его функциональность и позволят связываться с другими компонентами, необходимо использование *контейнера* — средства среды времени выполнения Java EE. В качестве примера служб, предоставляемых контейнерами для компонент, можно упомянуть следующие: Java API для транзакций, обработка JSON, службы безопасности, внедрение зависимостей и т. д. Взаимодействие между контейнерами осуществляется при помощи различных протоколов, например HTTP или JDBC. Контейнеры позволяют разработчику сосредоточиться на бизнес-логике приложения, вместо того, чтобы тратить время на технические проблемы, традиционно возникающие в корпоративных приложениях. На рис. 1 продемонстрирована

взаимосвязь четырех основных контейнеров в Java EE.

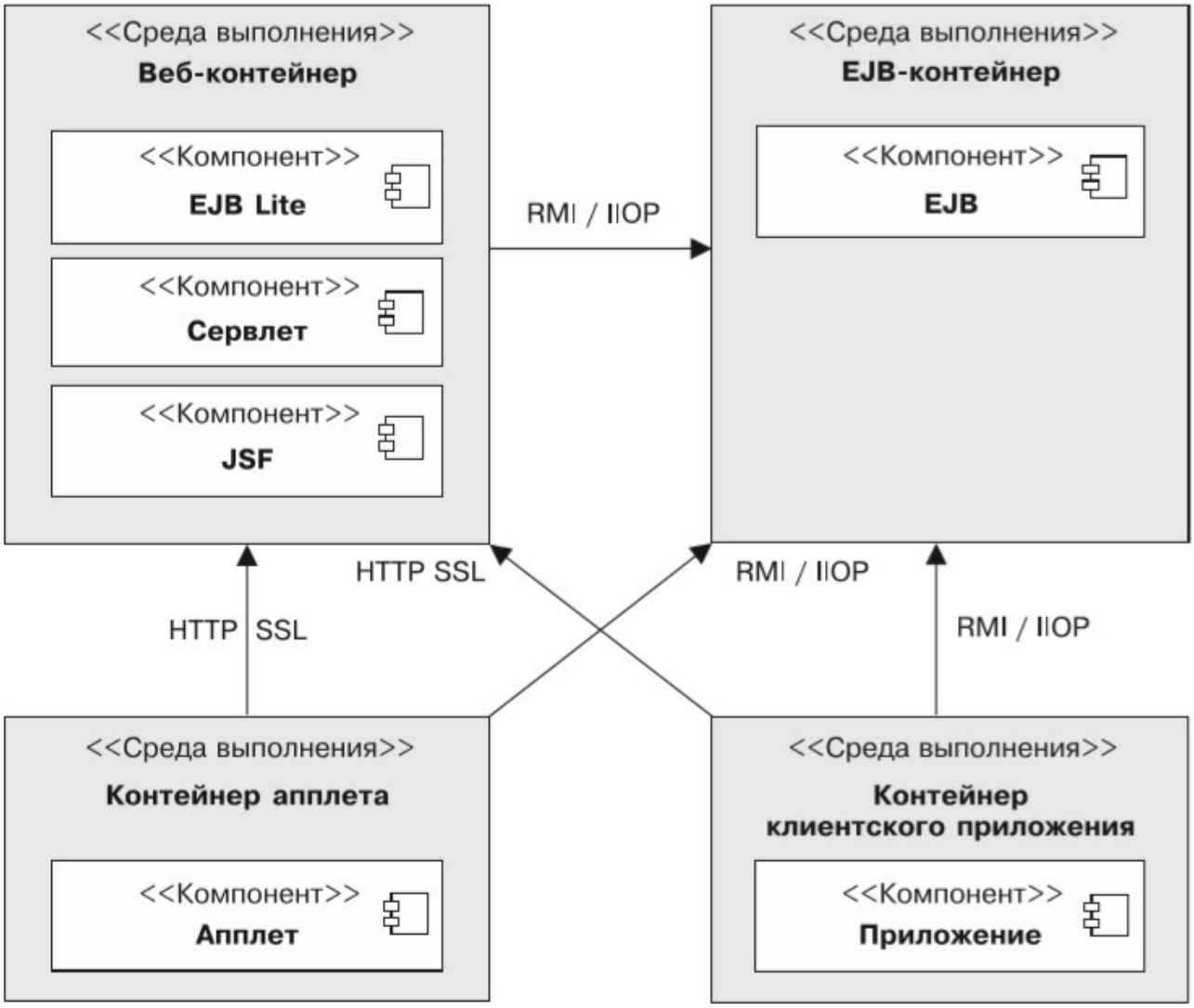

Рис.1. Стандартные контейнеры Java EE. [6]

Использование компонентов в контейнере подразумевает их упаковку в архивы формата jar. После чего в контейнере происходит их развертывание. Для взаимосвязи контейнера и определенного компонента нужны аннотации и/или дескрипторы развертывания. Аннотации — это метаданные, помогающие компилятору (или контейнеру) обрабатывать код. Например, аннотация *@Override* сообщает компилятору, что данный метод переопределен. Если метод не будет найден в родительском классе или интерфейсе, то это приведет к ошибке компиляции. Вместо аннотаций можно использовать дескриптор развертывания. Это файл XML-конфигуратора, который выполняет те же функции, что и аннотации. Примером такого файла может служить *web.xml* для веб-контейнера. Преимущество дескрипторов развертывания в том, что при внесении изменений, нужно менять отдельный файл, вместо изменений в исходном коде, который затем нужно будет перекомпилировать. С другой стороны, аннотации — это конструкции языка Java, и они занимают гораздо меньше места, чем код на XML. На рис. 2 показано, как располагаются архивы внутри контейнеров, включая дескрипторы

развертывания.

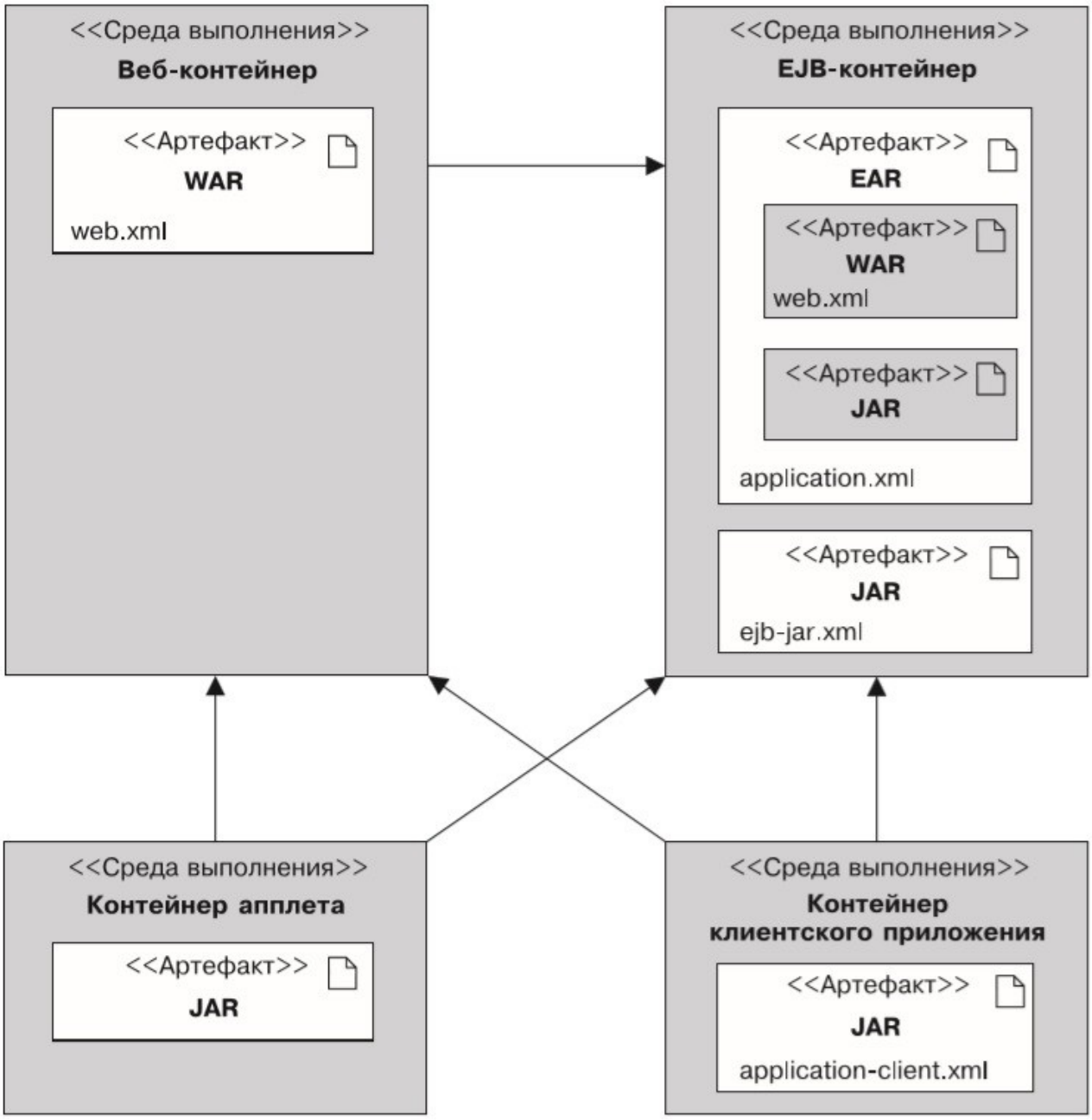

Рис.2. Архивы в контейнерах. [6]

Выводы по главе 1

- Программный комплекс distolymp можно разделить на две части: клиентскую и серверную, причем последняя состоит из PHP-подсистемы, отвечающей за логику, и СУБД, отвечающей за управление базой данных.

- Основной особенностью клиентской части является наличие проигрывателя BarsicPlayer, который обладает широкими возможностями по работе с виртуальными моделями и упрощает выполнение математических операций для пользователя.

- Крупные серверные приложения на Java определяются стандартами Java EE. Как правило, они состоят из множества слабосвязанных компонент, которые распределены на группы и после архивирования расположены внутри контейнеров. Контейнеры обеспечивают работу таких компонент. В этом им помогают аннотации или дескрипторы развертывания.

2. Технологии, необходимые для Java-версии программного комплекса distolymp

2.1 Контейнер сервлетов Apache Tomcat

Как было упомянуто в первой главе, работа с компонентами Java EE осуществляется при помощи контейнеров. Одним из важных компонентов серверной части является сервлет, позволяющий обрабатывать запросы клиентов. Сами по себе сервлеты, расположенные на сервере, бесполезны. Для работы с ними необходимо использование контейнера сервлетов, который будет осуществляеть их жизненный цикл, состоящий из трех этапов: инициализация, обслуживание клиентского запроса и уничтожение.

Одним из наиболее известных контейнеров сервлетов является Apache Tomcat, который разрабатывается компанией Apache Software Foundation. Данный контейнер сервлетов может работать и как самостоятельный сервис, но чаще используется в связке с другими веб-серверами, например Apache HTTP-сервер. Одна из главных причин — это распределение нагрузки, в частности, на внешний веб-сервер можно перенести работу со статическим контентом, шифрованием (при работе с SSL), настройками безопасности, кешированием и т. д. Для связывания Apache Tomcat с Apache HTTP-сервером, как правило, используют mod_jk — специальный модуль, который использует протокол AJP (Apache JServ Protocol) для перенаправления запросов с внешнего сервера на Apache Tomcat. На рис. 3 продемонстрирована возможная структура такой системы. Вопросы связывания не рассматривались в данной работе, поэтому здесь нет подробного описания настройки конфигурационных файлов.

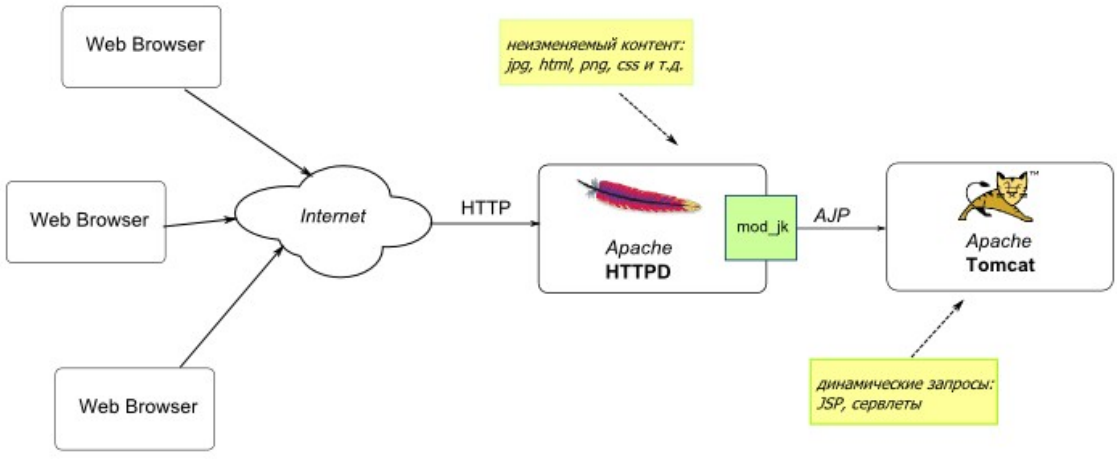

Рис.3. Структура связки Apache Tomcat с Apache HTTPD. [7]

Рассмотрим кратко работу Apache Tomcat с точки зрения внутренней архитектуры. На

рис. 4 приведена общая схема частей, из которых состоит данный контейнер сервлетов. Существуют так называемые top level components, к которым относятся *server* и *servise.* Server — это объект singleton (для данного процесса может существовать только один экземпляр данного объекта), который создается JVM (Java Virtual Machine) и связывается с данным Apache Tomcat. Каждый такой объект настраивается при помощи конфигурационного файла server.xml и содержит набор services, которые предоставляют список компонент более низкого уровня для обработки запросов.

Каждый service имеет группу *connectors*, которые нужны для передачи запросов от клиента к той части контейнера, что отвечает за их обработку. Каждый connector связан со своим протоколом и фактически является внешним интерфейсом для Apache Tomcat. Connector передает запрос процессу из пула процессов. Процесс запускает *engine,* единственный для данного service. Engine может включать в себя следующие сущности: *host, context, wrapper*. Здесь host отвечает за возможность при запросе к разным доменам обращаться к одному и тому же engine. С другой стороны, один и тот же host может отвечать за разные веб-приложения, каждое из которых соответствует своему context. С точки зрения сервера, такое приложение размещается на нем в формате WAR-архива. В нем должен быть дескриптор развертывания web.xml, необходимый данному context. После развертывания WAR-архив преобразуется в каталог, структуру которого определяет context. В этом каталоге, в частности, содержатся сервлеты. Каждый сервлет обернут в компонент *wrapper,* который отвечает за его жизненный цикл.

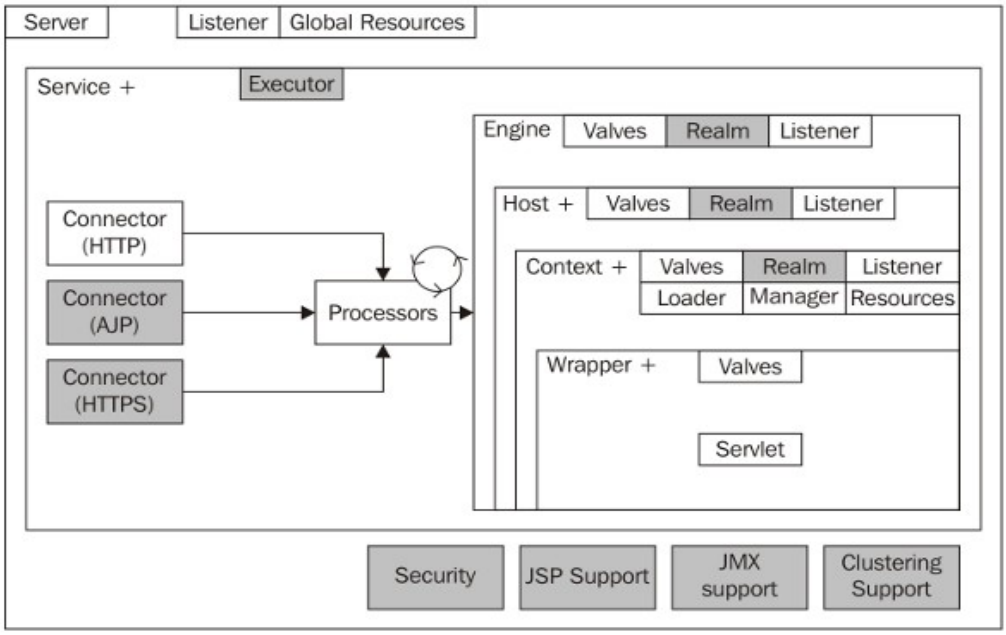

Рис.4. Внутренняя архитектура Apache Tomcat. [8]

2.2 Платформа автоматизации сборки Maven

В современном программировании крупные проекты включают в себя огромное количество разнообразных файлов, которые нетривиально связанны между собой. Таким образом этапы компиляции, сборки, проведения тестов, разворачивания программы сильно усложняются, если их не автоматизировать. Для этого используют различные инструменты, такие как Cmake, Ant, Maven, Gradle и т. д. В нашей работе было решено использовать платформу Maven, так как она предназначена для сборки проектов, написанных на Java для решения типовых задач. Maven реализует декларативную сборку проекта, это означает, что сценарии сборки не содержат последовательных команд, а лишь описывают исходные данные и то, что должно получится в конце. Среди других плюсов данной платформы можно отметить независимость от операционной системы, хорошую интеграцию с основными IDE для java, удобное управление зависимостями [9].

При создании проекта с использованием maven, необходимо выбрать среди шаблонов (архетипов) проекта нужный для написания конкретного вида приложения. Такой выбор определит структуру директорий в проекте, которую maven создаст по умолчанию. В частности, нам необходим архетип maven-archetype-webapp. На рис. 5 приведена примерная структура для этого архетипа. На рисунке видно уже созданные стандартные файлы для веб-приложений java: web.xml и index.xhtml.

```
|-- pom.xml
|-- src
|   `-- main
|       |-- java
|       |   `-- org
|       |       `-- mycompany
|       |           `-- AccountBean.java
|       `-- webapp
|           |-- WEB-INF
|           |   |-- beans.xml
|           |   |-- web.xml
|           |   `-- weblogic.xml
|           |-- css
|           |   `-- bootstrap.css
|           |-- index.xhtml
|           `-- template.xhtml
```

Рис.5. Структура проекта для maven-archetype-webapp. [9]

Особенно важным для maven является файл pom.xml, так как именно он описывает проект. Вообще говоря, таких файлов может быть несколько, если проект особенно сложный, но в нашем случае этого не потребовалось. В pom.xml происходит идентификация проекта по значениям параметров: groupId — название организации и artifactId — название проекта. Помимо этого, существует параметр version, который определяет версию проекта. Основу pom.xml составляет список зависимостей (библиотек), которые перечислены внутри тега

<dependencies>. Благодаря этой информации, maven способен в нужный момент подключить библиотеку к проекту. Для того, чтобы платформа могла понять, какая библиотека нужна, внутри зависимости используют также три параметра: groupId, artifactId, version. Библиотеки, которые используются при разработке проектов, расположены в репозиториях, к которым и обращается maven. Помимо общеизвестных репозиториев, можно работать с локальными, после того, как они будут указаны в pom.xml. Также в pom.xml можно указывать расположение ресурсов, которые необходимы при сборке, если такое расположение не соответствует значениям по умолчанию.

В процессе сборки проект проходит несколько фаз [10]. Некоторые могут быть пропущены, что не повлияет на работоспособность собранного проекта. Ниже приведены основные фазы сборки:

- *compile* — отвечает за процесс компиляции;
- *test* — проведение unit testing для проверки отдельных модулей программы;
- package — процесс упаковки файлов проекта в специальный архив (например jar или war);
- *integration-test* — тестирование более высокого уровня, в отличие от unit testing, тестируются не отдельные модули, а их совокупность;
- *install* — копирование архива проекта в локальный репозиторий;
- *deploy* — размещение на удаленном репозитории, доступ к которому могу иметь другие разработчики.

2.3 Программный комплекс Spring для платформы Java

Разработка крупных проектов связана со сложностями не только на этапе сборки, но и на этапе разработки. Многочисленные связи между частями проекта, жизненные циклы объектов, сложное поведение программы в стандартных ситуациях — все эти проблемы тяжело решить, используя традиционные методы разработки программ. Поэтому в промышленном программировании возникли специальные программные платформы, получившие название «фреймворки», которые были способны определять структуру и поведение будущего проекта. Один из наиболее известных фреймворков, ориентированных на разработку промышленных java-приложений, — Spring. Популярность вызвана такими факторами, как: огромное количество дополнений для решения различных задач, наличие хорошей документации, возможность создания приложений из слабосвязанных компонентов. Фактически, Spring выступила наследником спецификации EJB (Enterprise JavaBeans), но со значительно упрощенным процессом создания компонентов.

Можно перечислить четыре основные стратегии данной платформы [11]:

- использование POJO (Plain Old Java Object), за счет чего достигается легковесность приложений;
- внедрение зависимостей и широкое применение интерфейсов, что приводит к слабой связанности;
- концепция декларативного программирования (при помощи использования аспектов и общепринятых соглашений);
- стремление сократить исходный код через шаблоны и аспекты.

Основные понятия Spring

Рассмотрим два основных понятия, которые лежат в основе Spring. Это *внедрение зависимостей (DI — dependency injection)* и *аспектно-ориентированное программирование (AOP — aspect-oriented programming).*

Внедрение зависимостей подразумевает, что объект некого класса не должен заботиться о зависимостях между ним и объектами других классов, он может лишь предоставлять методы, которые позволяют ему получить необходимые ссылки, олицетворяющие данные зависимости. За такие связи между объектами отвечает специальный механизм, в частности, в платформах создается так называемый IoC-контейнер (Inversion of Control), внутри которого происходит работа приложения. Такой контейнер будет отвечать за установление зависимостей между созданными внутри него объектами. Это позволяет добиться слабой связи между компонентами программы и упрощает их тестирование, но усложняет восприятие кода из-за наличия дополнительных конфигурационных файлов контейнера.

Аспектно-ориентированное программирование — парадигма, позволяющая отделить функциональность при разделении проекта на меньшие части. В любом крупном проекте существуют системные службы, которые решают задачи безопасности, ведения логов, управление транзакциями и т. д. Другими словами, это *сквозные (scattered)* задачи, которые затрагивают одновременно множество частей программы. Причем отдельные объекты, имеющие свою собственную функциональность, оказываются нагружены такими сквозными задачами. Это увеличивает программный код, усложняет его и значительно уменьшает гибкость проекта. В результате применения AOP такие службы отделяются от частей проекта и превращаются в аспекты (обертки над частями приложения). Пример подобной системы представлен на рис. 6, где в центре рисунка представлены компоненты проекта, обернутые в модули, которые решают задачи системных служб.

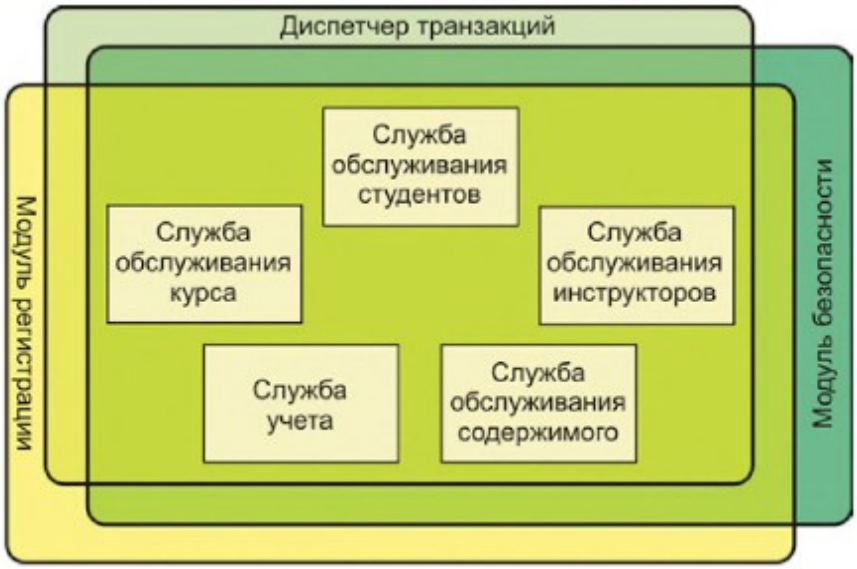

Рис.6. Пример системы с использование AOP. [11]

Как было сказано ранее, в Spring существуют контейнеры, внутри которых осуществляется работа частей приложения. Различают два типа подобных контейнеров: *фабрики компонентов* и *контекст приложений*. Первый тип является более простым, поддерживает меньше функций и используется для написания приложений на маломощные устройства. Второй тип, помимо загрузки, связывания и конфигурации компонентов (объектов классов, которые написаны по определенным правилам), также осуществляет работу с прикладными службами платформы (получение текстовых сообщений от ряда файлов).

Специальный объект какой-то из разновидностей контекста приложений выполняет поиск контекста приложения либо в XML-файлах, либо изучает аннотации, тем самым, в процессе его работы создаются все необходимые компоненты, затем происходит их настройка, а, после окончания работы, их уничтожение.

Стоит заметить, что сама платформа Spring состоит из ряда модулей, каждый из которых отвечает за свои классы задач. На рис. 7 изображены основные модули Spring, часть из которых была применена в данной работе. Более подробно об их использовании будет рассказано в 3 главе.

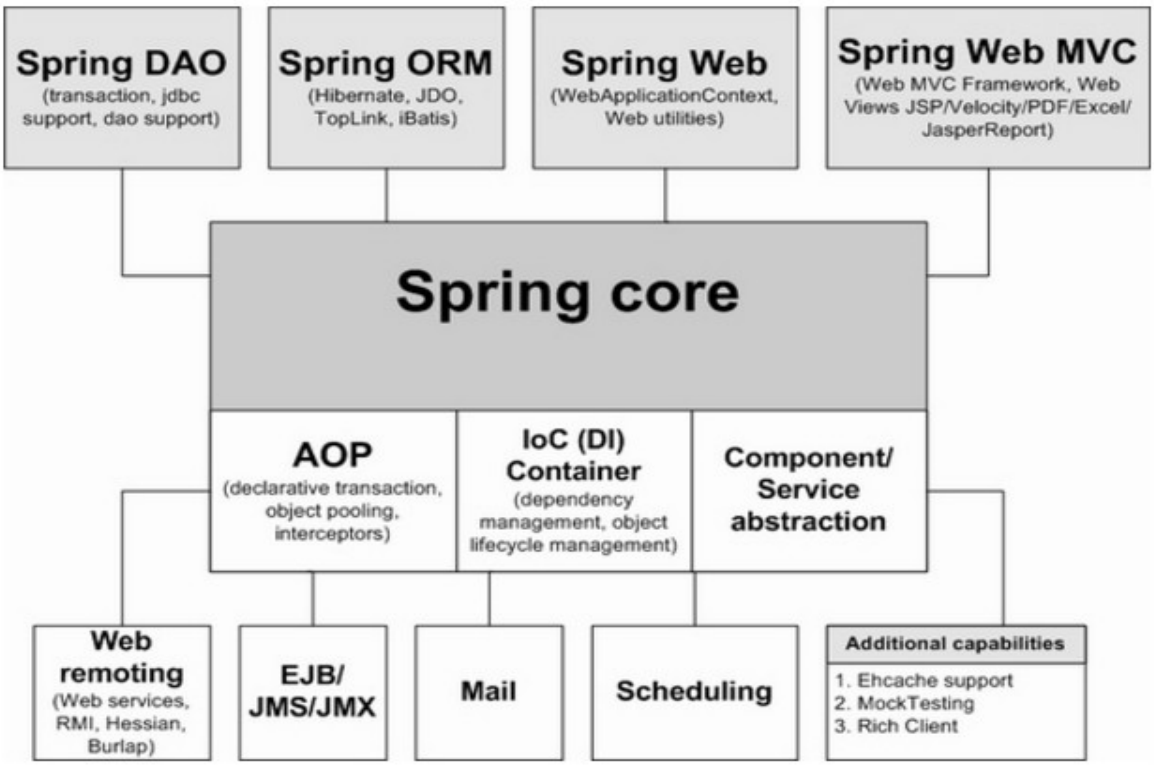

Рис.7. Модули Spring. [12]

2.4 Платформа Hibernate для решения задач ORM

В процессе работы с базами данных возникает задача создания похожей структуры в терминах объектно-ориентированного программирования, в которых описывается основной код приложения. Такого рода задачи имеют название: задачи ORM (Object-Relational Mapping). Применение подобных идей очень сильно уменьшает код, отвечающий за взаимодействие с базами данных, делая его более прозрачным. Существует ряд технологий, помогающих разработчикам решать подобные проблемы. В нашей работе используется платформа Hibernate, которая обладает широким функционалом для описания взаимодействия кода бизнес-логики с базами данных. Фактически, создается виртуальная база данных (структура классов, повторяющая таблицы базы данных и связи между ними), объекты которой играют роль буферов, в которые можно загружать данные из базы или данные для записи в базу. Причем с ними можно работать как с обычными объектами в Java в рамках кода, не связанного с базами данных.

Основным интерфейсом в работе между Java и Hibernate является интерфейс Session. Объект данного типа осуществляет одну CRUD (Create Read Update Delete) операцию между приложением и базой данных. Такой объект является синглтоном и работает с объектами-сущностями из виртуальной базы данных. Любой объект-сущность может находиться в трех состояниях [13]:

- transient object — объект, не связанный с сессией и не был связан до этого (поле Id не заполнено), он не относится к конкретной записи в базе данных и в будущем может быть там сохранен;
- persistent object — состояние, при котором возможно взаимодействие с базой данных, объект присоединяется для этого к сессии, любое изменение такого объекта сказывается на привязанной к нему записи в базе данных;
- detached object — объект, который был отсоединен от сессии, но в будущем может быть присоединен к новой, причем запись, связанная с ним раньше, уже может быть изменена;

Существуют различные методы, отвечающие за переходы между данными состояниями. Например, *persist* отвечает за переход из первого состояние во второе (при этом объект сохраняется в базе данных), а обратный переход может быть выполнен методом *delete*. Для перехода от третьего состояния ко второму используют метод *update*. Важно заметить, что все эти методы по-разному обходятся с полем Id и используют разные команды SQL (Structured Query Language) для запросов к базе данных. Все эти механизмы позволяют корректно работать с базами данных и не производить с ними тех действий, последствий которых уже нельзя будет исправить.

2.5 Платформа Mockito для unit-тестирования

Достаточно важной частью любого крупного проекта являются тесты. Однако при тестировании необходимо учитывать кучу нюансов из-за большого числа компонентов такого проекта. Если разработчик хочет протестировать только одну часть такого проекта, то он должен использовать связи этой части с другими, что не всегда разумно. Поэтому при таком тестировании часто используют mock-объекты — фиктивные объекты неких интерфейсов, задача которых правильно воспроизводить ожидаемое поведение. Такой объект не просто помогает симулировать поведение всех объектов, связанных с тестируемым, но и помогает собирать статистику о поведении системы. Таким образом, если в какой-то момент работы программы она не совершила нужного действия, mock-объект сообщит об этом. Классический пример — это проверка вызовов методов. Если объект должен на каком-то этапе вызвать конкретный метод, то такое действие будет зафиксировано. Можно сказать, что mock-объекты являются умной реализацией заглушки (нечто подменяющее действие реального метода или объекта).

Существует много инструментов, помогающих в тестировании. Автором была выбрана платформа Mockito, как одно из популярных решений в области тестов в Java. Данная платформа предоставляет множество удобных инструментов для написания качественных

тестов и, благодаря ее популярности, можно найти много примеров и обучающих материалов по её использованию.

Выводы по главе 2

- Проведен анализ необходимых программ и платформ для создания пробной версии веб-приложения на Java, которая в дальнейшем заменит текущую php-версию distolymp.

- Для корректной работы сервлетов, которые являются важной частью веб-приложений на Java, необходим контейнер сервлетов. Автор выбрал Apache Tomcat, как наиболее распространенное и надежное решение данной проблемы.

- Автор посчитал целесообразным использование ряда платформ для решения задач сборки, создания кода и его тестирования. Такой подход значительно упрощает дальнейшую работу над проектом. Кроме того, подход удовлетворяет современным требованиям на рынке веб-приложений.

3. Структура проекта

MVC (Model View Controller) — основной паттерн проектирования, определяющий работу веб-приложений. Он позволяет разделить приложение на три части, которые отвечают за свои задачи. Controller управляет приходящими от клиента запросами, Model формирует ответ, удовлетворяющий требованию запроса, View отображает ответ в нужном для клиента формате. На рис. 8 видно из каких частей состоит типичное приложение Spring MVC.

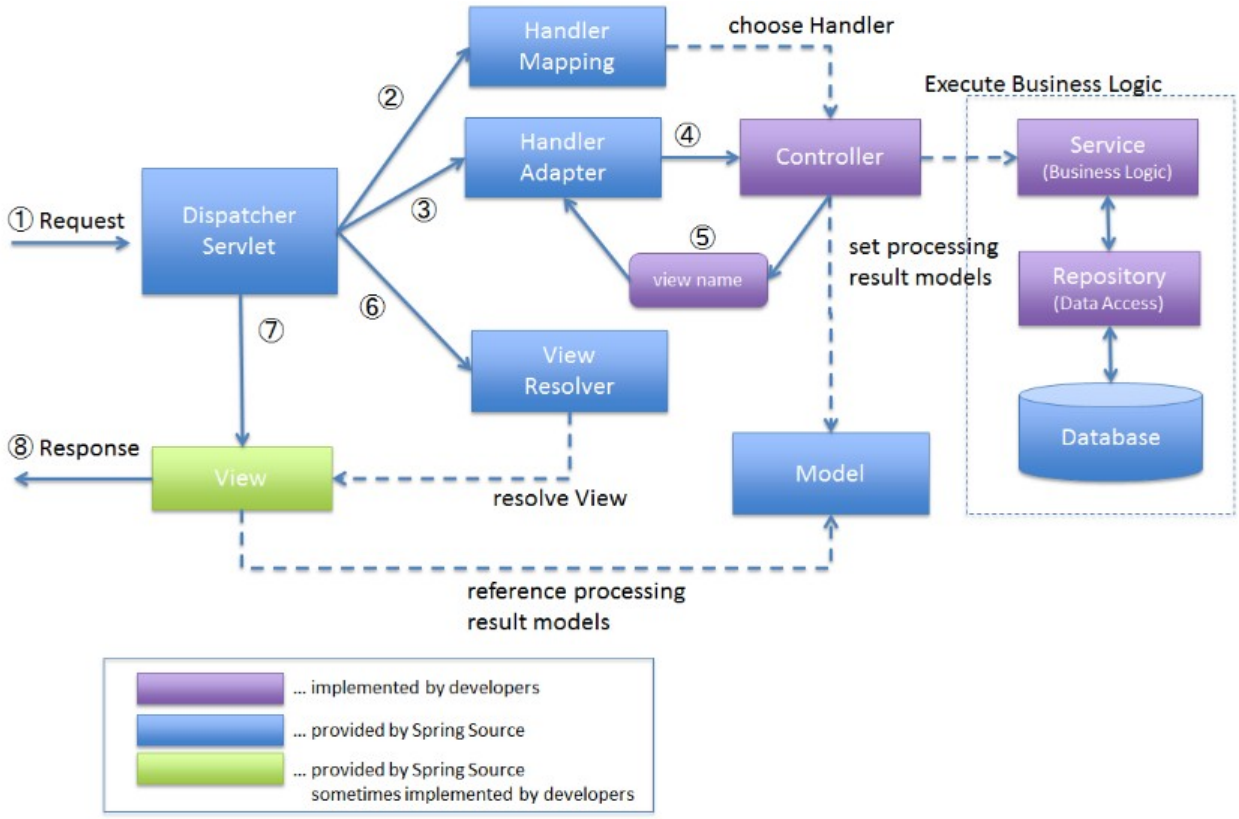

Рис.8. Структура Spring MVC проекта. [14]

Цифры указывают очередность действий внутри такого приложения после того, как к нему приходит запрос. Какие-то части могут отсутствовать, но общая схема от этого не изменится. На рис. 9 показана структура разработанного приложения. В дальнейшем само приложение будет претерпевать изменения, но его структура останется соответствующей рис. 9. Ниже будут подробно разобраны все те части схемы, реализованные в нашем проекте.

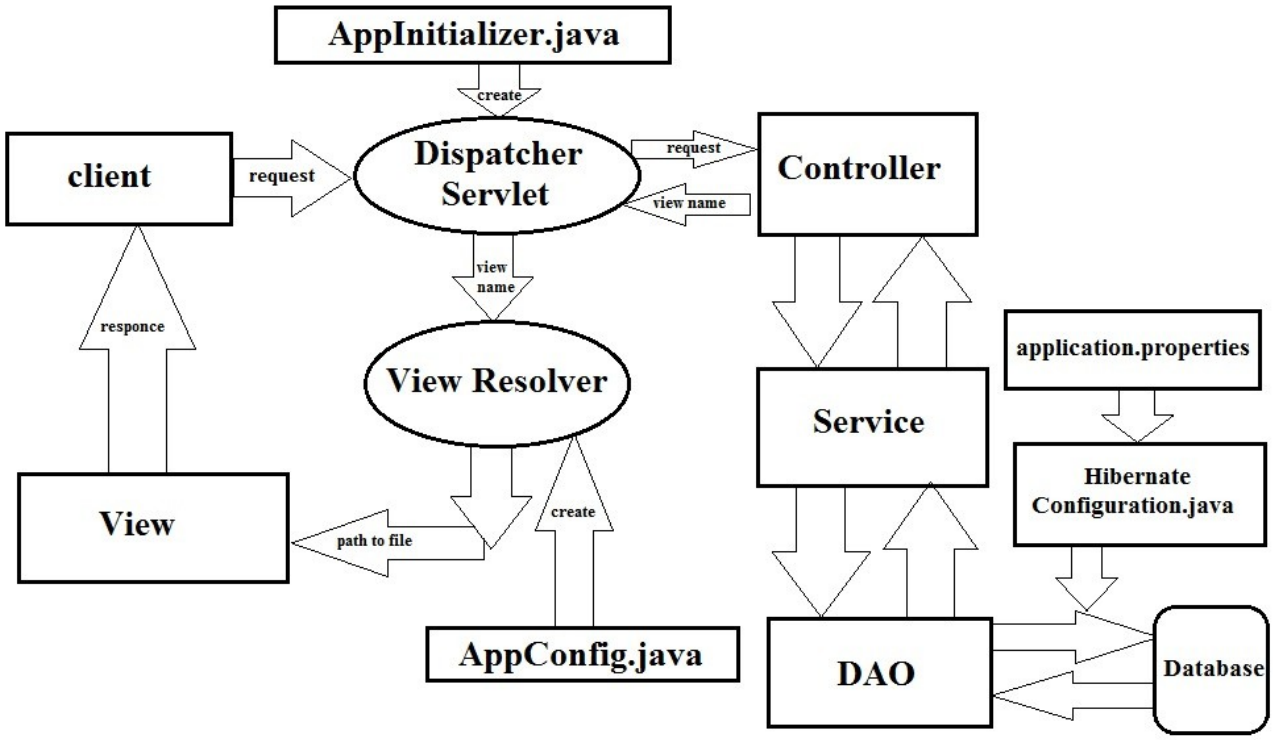

Рис.9. Структура приложения для регистрации пользователей.

3.1 Конфигурационные файлы

Здесь мы перечислим основные файлы, отвечающие не за логику проекта, а за некоторые его настройки.

Файл pom.xml

Начнем с уже упомянутого во второй главе основного файла платформы maven — pom.xml. Мы будем указывать расположение всех файлов относительно пути от основной папки проекта после распаковки WAR-архива. Сам проект был назван DBProject (от database, так как эта версия расширяла изначальный вариант для работы с базами данных). И файл pom.xml находится по адресу «.../», то есть в корневой папке. Там же находится файл nb-configuration.xml, сообщающий о том, что разработка проекта велась в IDE Netbeans.

Структура и задачи pom.xml уже были описаны, поэтому здесь мы перечислим groupId (идентификаторы неких репозиториев, откуда брались основные библиотеки), указанные в этом файле:

- javax;
- org.springframework;
- org.hibernate;
- mysql-connector-java;

- javax.validation;
- joda-time;
- org.jadira.usertype;
- javax.servlet;
- javax.servlet.jsp;
- com.fasterxml.jackson.core;
- org.testing;
- org.mockito;
- com.h2database;
- dbunit.

Удобство pom.xml в том, что в случае добавления новой библиотеки или новой версии уже добавленной библиотеки достаточно внести изменения в один xml-файл. Стоит заметить, что часть библиотек, например, из репозиториев joda-time (удобная работа с параметром «время» в Java) или com.fasterxml.jackson.core (работа с форматом данных JSON), подключена для дальнейшего развития проекта и мало использовалась в текущей работе. Люди, которые продолжат работать над проектом, смогут сами решить, необходимо ли их использование.

Файлы properties

Следующие важные файлы в проекте — это файлы с расширением properties. Они находятся по адресу «*/src/main/resources»* и выполняют задачу хранения конфигурационных параметров проекта. Весьма важно, что это текстовые файлы, и изменения в них не требуют новой сборки проекта. Информация хранится в виде пар «ключ»-«значение», и ключ потом может использоваться в любом файле проекта. В DBProject имеются два подобных файла. Первый называется application.properties и содержит информацию, необходимую для работы с базой данных (например, пароль или логин). Второй файл называется messages.properties, и он предназначен для хранения сообщений, которые будут выводится при различных ошибках. Данный файл также создан для использования в будущем и может быть удален при развитии проекта.

Файлы директории configuration

Наконец наиболее важные файлы для логики проекта расположены по адресу «*/src/main/java/com/mycompany/dbproject/spring/configuration»*. Здесь находятся три файла: AppConfig.java, AppInitializer.java и HibernateConfiguration.java.

AppInitializer.java необходим для регистрации DispatcherServlet при старте контейнера. На

рис. 9 это показано стрелкой. Файл является аналогом дескриптора развертывания web.xml. Именно с него начинается работа приложения, как с класса, расширяющего интерфейс AbstractAnnotationConfigDispatcherServletInitializer. Методы getRootConfigClasses и getServletConfigClasses возвращают классы, которые будут определять контекст для создания всех необходимых компонентов. Распределение конфигурационных классов между ними диктуется структурой крупного проекта. В нашем случае у нас только один подобный класс AppConfig.java, и его возвращает метод getRootConfigClasses. Третий метод в AppInitializer.java — это getServletMappings. Он определяет URL, при обращении к которому будет вызван DispatcherServlet. В нашем случае — это «/» (запрос относительно названия WAR-архива, то есть «DBProject/»).

Перейдем теперь к AppConfig.java. В первую очередь он упрощает доступ к статическим ресурсам, а именно файлам css, javascript и изображениям (расположенными по адресу «*/src/main/webapp/resources»* в соответствующих папках*)*. Вместо пути «*/resources/css»* можно использовать «*/css» (*аналогично для других ресурсов). Помимо этого, в классе создаются компоненты *viewResolver* и *messageSource* (на рис. 9 показана роль viewResolver в структуре проекта). Первый отвечает за формирование адреса, где программа будет искать файл, необходимый для View, который будет возвращен клиенту. В классическом варианте контроллер после обработки запроса возвращает строчку, которая является именем файла представления. Задача viewResolver (арбитра представлений) добавить к ней префикс (путь до файла) и суффикс (расширение файла). В нашем случае префикс — «/WEB-INF/views/», а суффикс — «.jsp». Второй компонент необходим для механизма вывода сообщений. Как уже было сказано, в нашей работе он фактически не применялся.

Последний файл HibernateConfiguration.java создает все необходимое для работы Hibernate с базами данных. Где он расположен в структуре проекта видно на рис. 9. Стоит отметить, что в начале файла происходит подключение файла properties за счет аннотации @PropertySource(value={ "classpath:application.properties" }). Затем создается ряд компонентов. Напомним, что все CRUD операции выполняются при помощи сессий, которые взаимодействуют с объектами-сущностями. Таким образом, нужен компонент sessionFactory, фактически, фабрика сессий, возвращающая объект-синглтон. Для того, чтобы сессии работали с базой данных, они должны подключиться к ней. Вся необходимая информация предоставляется компонентом dataSource, который получает ее, в свою очередь из properties-файла. Помимо информации для подключения, существует информация об устройстве самой базы данных. Её предоставляет компонент hibernateProperties. Последний компонент HibernateTransactionManager создает транзакцию. Чтобы привязать транзакцию к сессии, данный компонент требует на вход объект класса SessionFactory. Аннотация @Autowired перед

компонентом сообщает, что данный объект будет автоматически предоставлен HibernateTransactionManager, если он уже будет создан в контексте данного приложения.

3.2 Слой Model

Данный слой является виртуальным аналогом базы данных, состоящим из объектов-сущностей. Автор в процессе работы над проектом воссоздал часть базы данных, которая используется в distolymp. Было создано 14 таблиц. Следует заметить, что при создании таблицы passports возникла ошибка при создании следующих столбцов: *where, who, when*. Версия MySQL Workbench, использованная для создания базы данных, сообщила, что такие названия являются зарезервированными словами для SQL. Поэтому было принято решение назвать эти столбцы: *where_passport, who_passport, when_passport*. Таким образом, каждой таблице соответствует класс с аналогичным названием в директории «*/src/main/java/com/mycompany/dbproject/spring/model*» (отметим сразу, что все следующие слои, кроме View расположены также по адресу «*/src/main/java/com/mycompany/dbproject/spring/*» в директориях с соответствующими названиями). В начале класса аннотация @Entity сообщает, что это класс слоя Model, и он является сущностью. Следующая за ней аннотация @Table(name="table_name)") уточняет название соответствующей таблицы в базе данных. Для того, чтобы воспроизвести структуру данной таблицы, каждому столбцу ставится в соответствие поле данного класса с похожим название. Специальные аннотации позволяют наиболее точно охарактеризовать данный столбец. Приведем часть аннотаций, которые были использованы в слое Model:

- @Column(name = "id_division", nullable = false) — аннотация сообщает, что поле имеет аналог в виде столбца с именем id_division в данной таблице, и что поле не может иметь значение null;

- @Id — ставится перед полем, которое соответствует первичному ключу в данной таблице;

- @Size — определяет максимальный и минимальный размер значения поля;

- @GeneratedValue(strategy = GenerationType.IDENTITY) — как известно, первичный ключ должен иметь уникальное значение и автоматически генерироваться, поэтому данная аннотация передает ответственность за это поле базе данных;

- @OneToMany (mappedBy = "idSchool", fetch = FetchType.LAZY, cascade = CascadeType.ALL) — данная аннотация решает проблему так называемого «Foreign

Key» (ключа, который связывает столбец одной таблицы с множеством записей в другой). Такая аннотация ставится перед списком объектов другой таблицы, связывая его поле, указанное в mappedBy. Параметр fetch здесь определяет ленивую загрузку, данный список заполнится, только если это понадобится. Параметр cascade означает, что в случае, например, удаления объекта с данным Foreign Key, удалятся (каскадом) и все объекты из другой таблицы, которая была с ним связана. Пусть для определенного города существует много школ (они будут связанны через ключ idTown), если же удалить этот город, то очевидно, что и все школы из него должны быть удалены, как и все ученики данных школ — это и есть каскадная операция.

Важно, что все поля и списки имеют свои сеттеры и геттеры (методы, позволяющие записать значение в поле или считать его из поля).

3.3 Слой DAO

Структура класса AbstractDao, как следствие структуры классов слоя DAO

В проекте, где имеется взаимодействие с базами данных, необходимо наличие слоя, который отвечает только за подобное взаимодействие. Поэтому основные функции слоя DAO (Data Access Object) — это осуществление транзакций по необходимым запросам. Причем обработка полученной информации ложится на другие слои, хотя первичная фильтрация такой информации может проходить уже при составлении запроса. На рис. 9 была показана связь DAO с базой данных и другими слоями.

Аннотация @Repository сообщает нам, что данный класс принадлежит слою DAO. Структура слоя строится на одном абстрактном классе и множестве наследников. Каждый наследник расширяет свой специфичный интерфейс. Такая структура обусловлена рядом факторов. Во-первых, каждый класс-наследник осуществляет работу со своим классом из слоя Model. Это приводит к разным классам для объектов-сущностей в реализации методов слоя DAO. Такая проблема решается при помощи Generics (обобщенных данных), а именно в классе AbstractDao строится каркас класса слоя DAO. Используются обобщенные классы: PK (класс для объекта-ключа, по ключу можно получить объект из базы данных) и T (класс сущности). Во-вторых, для каждого класса слоя Model могут потребоваться свои специфичные методы при работе с базой данных, что будет отражено в соответствующем интерфейсе.

Помимо объявления Generics, в AbstractDao создается объект sessionFactory, которому автоматически присваивается значение. Метод getSession позволяет при помощи данного

объекта получить объект-сессию. После чего сессия используется в основных методах данного класса: *getByKey*, *persist*, *delete*. Для удобства работы также добавлен метод *createEntityCriteria* для создания объекта класса Criteria. Это довольно удобный механизм для работы с базами данных. На основании него строится запрос на языке SQL. Уже в классах-наследниках можно планировать работу запроса при помощи добавления к объекту criteria различных условий. Например, строка criteria.add(Restrictions.eq("idUser", idUser)) добавляет условие равенства поля idUser (это поле считывается с полученной информации из базы данных) и значения входного параметра некого метода (этот входной параметр также имеет название idUser, но никак не связан с базой данных), тем самым мы отсекаем все лишние столбцы, где idUser имеет другое значение.

Структура класса наследника AbstractDao

Для создания классов-наследников помимо связки с классом из слоя Model, автор также учел методы, существующие в php-подсистеме distolymp. Поэтому базовые методы: Load, Save, Delete, IsValidData были созданы под другими именами в классах-наследниках. На примере класса Users, мы имеем методы: findByIdUsers, saveUsers, deleteUsers, isValidData. Последний метод не был реализован, так как проблема, связанная с выводом ошибок, не решалась в этой работе. Автор предполагает, что данный метод будет либо модифицирован, либо удален в дальнейшем. В ряде классов слоя DAO были введены дополнительные методы, которые определялись задачами вывода информации для клиента. Например, в классе TownsDaoImp был введен метод findByNameTowns для решения задачи AJAX-запросов, когда пользователь пишет название города и поиск по id уже не работает. Автор предполагает, что в будущем будет построена более продуманная система подобных методов.

Еще один вопрос, возникший при разработки данного слоя, был связан с foreign key. Предполагается, что удаление, например, какого-то города из таблицы городов, приводит к удалению школ, связанных с ним. Поэтому при реализации метода deleteTowns, внутри запускается цепочка удаления школ, учеников и т. д.

3.4 Слой Service

Связью между слоем Dao и слоем Controller служит слой Service. Это видно на рис. 9. В слое Service должна содержаться вся бизнес-логика работы с полученными данными от слоя DAO перед возвращением их в Controller. Свидетельством того, что класс принадлежит именно этому слою, является аннотация @Service. Аннотация @Transactional предупреждает, что данный класс работает с транзакциями. Основная особенность транзакций заключается в том, что в случае ошибки, изменения, совершенные в процессе транзакции, отменяются, и все

возвращается к начальному состоянию.

В структуре слоя также используются интерфейсы и расширяющие их классы, но нет единого класса-родителя в лице абстрактного класса. В данной работе слой Service практически дублирует слой DAO, так как не было необходимости в бизнес-логике. Под дублированием подразумевается, что в каждом классе создается соответствующий объект класса из слоя DAO, и через него внутри методов класса слоя Service происходит вызов аналогичного метода для работы с базой данных.

3.5 Слой Controller

Слой Controller отвечает за получение запроса от клиента, получение необходимой информации через слой Service от базы данных и передачу этой информации слою View. Чтобы лучше представить себе это, следует посмотреть на рис. 9. Аннотация @Controller подтверждает, что класс действительно является контроллером. В работе был создан только один контроллер RegistrationController, который отвечает за задачу регистрации пользователей. В будущем, каждому модулю distolymp будет сопоставлен свой контроллер. Заметим, что аннотация @RequestMapping("/") сообщает, что запрос по URL «DBProject/» (здесь мы опустили домены) будет перенаправлен в данный контроллер. Обращение же по адресу «DBProject/location1/location2» может быть передано либо методу внутри данного контроллера (с аннотацией @RequestMapping("/location1/location2"), либо другому контроллеру. Структура подобных взаимосвязей должна быть продумана при дальнейшей разработке проекта.

Для работы с базой данных в контроллере создаются все необходимые объекты классов из слоя Service. Внутри методов также могут быть созданы объекты-сущности из слоя Model. Все эти объекты нужны для объекта model класса ModelMap, с которым и работают методы контроллера. Существует несколько типов подобных методов, но в этой работе использовались два: POST и GET. Они выполняют все необходимые действия при POST и GET запросах протокола HTTP.

Методы newUser и saveUser

Методы newUser и saveUser — два главных метода в данном контроллере, остальные методы были необходимы либо для отладки, либо для работы с AJAX. Начнем с GET метода newUser, который возвращает переменную строкового типа (название jsp-файла из слоя View). Основная задача метода — подготовить данные для View. Для этого создаются атрибуты объекта model. Эти атрибуты определяются данными, которые будут на странице регистрации в браузере, то есть, если там будут поля паспортных данных, то к объекту model добавляется атрибут passport, являющийся объектом класса Passports из слоя Model. Такой объект,

входящий в атрибут, содержит все необходимые поля, которые будут заполнены при взаимодействии пользователя в браузере со страницей регистрации (при заполнении страницы). На странице регистрации также присутствуют выпадающие списки, требующие данных. Таким образом, некоторые атрибуты объекта model являются объектами класса List. Например, пользователь в самом начале регистрации имеет возможность выбрать страну. Для того, чтобы выбор мог быть сделан, создается атрибут в виде списка объектов класса Countries. Таким образом, еще до момента отправления стартовой страницы клиенту, взаимодействие приложения с базой данных на сервере уже происходит.

После того, как клиент заполнил все поля и отправил свои данные на сервер, запрос перехватывает POST метод saveUser. В качестве входных параметров указаны объекты тех классов, которые будут необходимы для сохранения. Основная задача этого метода заключается в первичной проверке и сохранении данных. В данной работе удалось добиться сохранения паспортных данных и пользователя (данные записи расположены в разных таблицах и связанны через поля idPassport и idUser). Для корректной работы пришлось реализовать метод для получения id последней записи в таблице passports getLastIdPassport. Те поля, которые не заполняются клиентом в браузере, были заполнены произвольными значениями. Для корректной работы пришлось учесть, что два пользователя не могут иметь один и тот же логин или пароль, поэтому в заглушках для этих полей были использованы значения других полей. В дальнейшем нужно будет встроить генератор паролей и логинов в этом месте контроллера.

3.6 Слой View

Реализация слоя View с помощью JSP

Последним этапом в формировании ответа клиенту является слой View. Это видно на рис. 9. В нашей работе основу этого слоя представляет технология JSP (JavaServer Pages). Это одна из наиболее популярных технологий для подобного рода задач, которая основана на языке Java (в отличие, например, от Apache Velosity, синтаксис которого отличен от Java). Основная идея JSP состоит в том, что на основе статичного html-файла формируется jsp-файл с вставками Java кода, отвечающего за динамическую часть страницы. Такой файл на сервере преобразуется в html-файл, структура которого зависит от динамической части. Именно он и посылается клиенту.

Библиотека JSTL

Строки java-кода отделяются от статического контента при помощи символов «<% %>». Однако для удобства работы людей мало знакомых с Java создана библиотека JSTL (JavaServer

Pages Standard Tag Library). С ее помощью можно реализовывать простейшие динамические конструкции при помощи тегов, похожих на теги html. В данной работе были использованы теги с taglib prefix = “c”. Они выполняют простейшие задачи по выводу данных на страницу.

Как было упомянуто выше, от контроллера в данный jsp-файл передается объект model с набором атрибутом. JSTL позволяет прописать в теге <form> конкретный атрибут model и затем работать с полями атрибута. К сожалению, в случае нескольких атрибутов для одной формы, не предусмотрена возможность работать с ними одновременно. Данная проблема решается либо введением нескольких форм, что меняет существующую структуру страницы регистрации, либо поиском другого способа вывода информации. Автор принял решение использовать второй способ. Поэтому в jsp-файл также введен тег <spring:bind>, позволяющий использовать атрибуты с конкретным полем без ограничений. В листинге 1 приведена часть кода, в котором создается выпадающий список со странами.

Листинг 1. Код div-элемента, отображающего список стран

```
<div id="countries">
     <label for="country_id">Страна*:</label>
     <spring:bind path="country.name">
          <select  id="country_id"
          onchange="return setCountry(this.value)" class="wide" >
         <option value="-2" selected="selected" >выберите страну</option>
         <c:forEach items="${listCountries}" var="country">
              <option value="${country.idCountry}">${country.name}</option>
         </c:forEach>
         </select>
      </spring:bind>
</div>
```

JSTL-тег <c:forEach> последовательно проходит по значениям списка listCountries, присваивая переменной country значения из списка. Затем option присваиваются конкретные поля объекта country. Использование тега <spring:bind> позволяет также записывать в нужное поле атрибута значение, которое было введено клиентом в поле формы. Таким образом формируются данные для контроллера, которые отправляет клиент.

Проблемы с использованием запросов AJAX

В конце работы над страницей регистрации была предпринята попытка реализовать AJAX (Asynchronous JavaScript and XML)-запросы. Данная технология позволяет подгружать данные с сервера без повторного запроса html-страницы, что экономит время и ресурсы. На каждом слое проекта были созданы необходимые классы, но в процессе проверки работы автор столкнулся с ошибкой получения данных. Для обмена данными в AJAX часто используют специальный формат данных JSON (JavaScript Object Notation), который затем легко преобразовывать при помощи JavaScript в html-код. Была предпринята попытка формирования данных в контроллере именно в формате JSON перед возвращением клиенту. Но, по какой-то

причине, клиент получал html-файл вместо JSON. Метод контроллера, несмотря на нужные аннотации передавал управление слою View, который отсылал html-файл. Все вышесказанное удалось выяснить, выводя полученные данные в окно браузера при помощи метода alert. Решить данную проблему не удалось.

3.7 Развертывание готового проекта на production-сервере

После того, как программа заработал на ноутбуке под управлением ОС Windows 8, была поставлена новая задача: добиться работы программы на production-сервере distolymp. Для начала было необходимо установить необходимые программы для развертывания: JRE, Apache Tomcat и Maven. JRE (Java Runtime Environment) — минимальный набор средств для запуска приложений на Java, фактически, состоит из JVM и небольшого числа библиотек. Так как имеются два сервера distolymp: основной с ОС Debian 7.0 (Wheezy) и резервный с ОС Debian 6.0 (Squeeze), то было решено развернуть проект на резервном сервере.

Для решения задачи средствами VirtualBox была произведена эмуляция Debian 6.0, и в данной ОС была предпринята попытка установить необходимое ПО. В результате выяснилось, что данная версия ОС больше года не поддерживается разработчиками, из-за чего основные пакеты программ перенесены в архивный репозиторий, что не позволяет установить их стандартными инструментами Linux. Тогда было принято решение работать с Debian 7.0, где этой проблемы не было. После установки ПО на виртуальной машине, была проведена аналогичная операция на сервере. Оказалось, что для корректной работы Maven недостаточно JRE, поэтому был установлен JDK (Java Development Kit) — набор инструментов, позволяющий вести разработку на Java. Перед запуском основного проекта была проверена работа Apache Tomcat на тестовом проекте Spring. На рис. 10 изображен пример запуска Apache Tomcat на виртуальной машине, а на рис. 11 — запуск тестового примера.

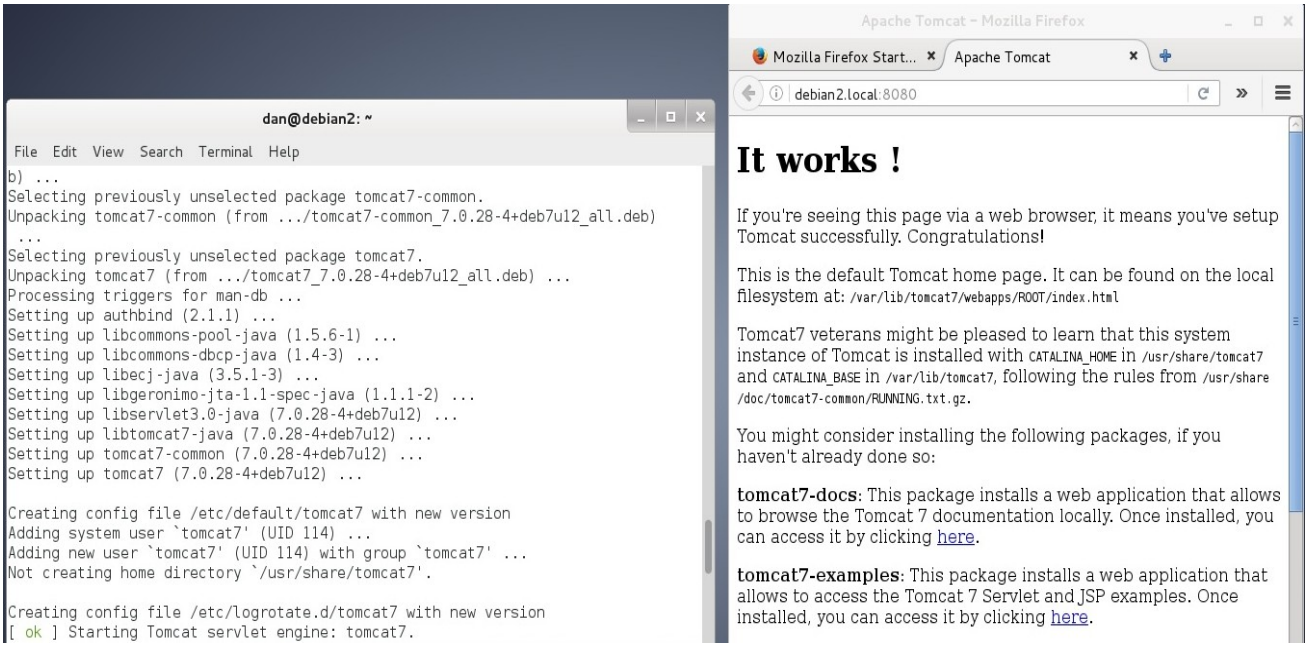

Рис.10. Запущенный Apache Tomcat на Debian 7.0.

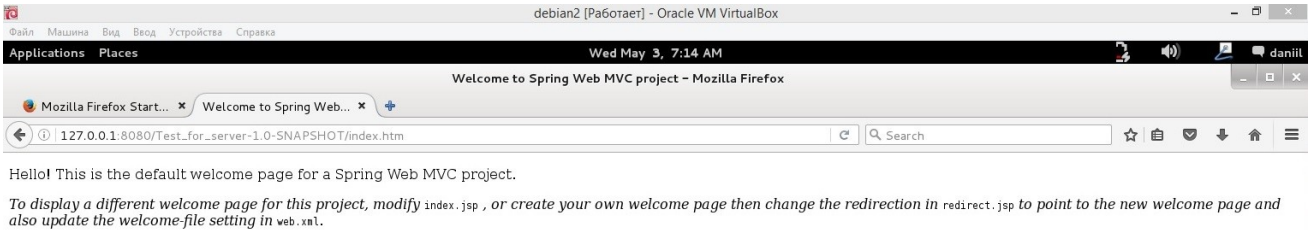

Рис.11. Работа тестового примера Spring на Apache Tomcat на Debian 7.0.

Перед запуском проекта регистрации пользователей на реальном сервере, на него была перенесена база данных с компьютера автора, а в файле application.properties был изменен пароль для базы данных. В процессе запуска возникла проблема с JDBC Driver. Для ее решения библиотека, отвечающая за данный драйвер (mysql-connector-java), была перенесена в папку библиотек Apache Tomcat, а в файл pom.xml было внесено исправление (в зависимость для mysql-connector-java была добавлена строка <scope>runtime</scope>) для того, чтобы данная библиотека подключалась во время выполнения программы, а не ее компиляции.

В версии проекта, которая запускалась на сервере, в контроллере была строка, позволяющая получать из пробной таблицы школ только школы из Петергофа. Для получения перед созданием страницы регистрации делался запрос к базе данных по слову «Петергоф». Это работало на компьютере автора, но привело к ошибке на сервере, хотя остальные запросы доходили до базы данных. При анализе ошибки было сделано предположение, что проблема состоит в различие кодировок в проекте и базе данных. Но эта версия не подтвердилась, после сравнения кодировок. Проблема с получением школ из Петергофа не была решена, но после того, как эта строка была удалена, проект заработал. Все действия, выполненные на сервере,

были занесены в специальный файл; помимо команд, в этот файл также добавлены ошибки, с которыми столкнулся автор, и способы их решения. На рис. 12 продемонстрирована работа данного проекта на production-сервере.

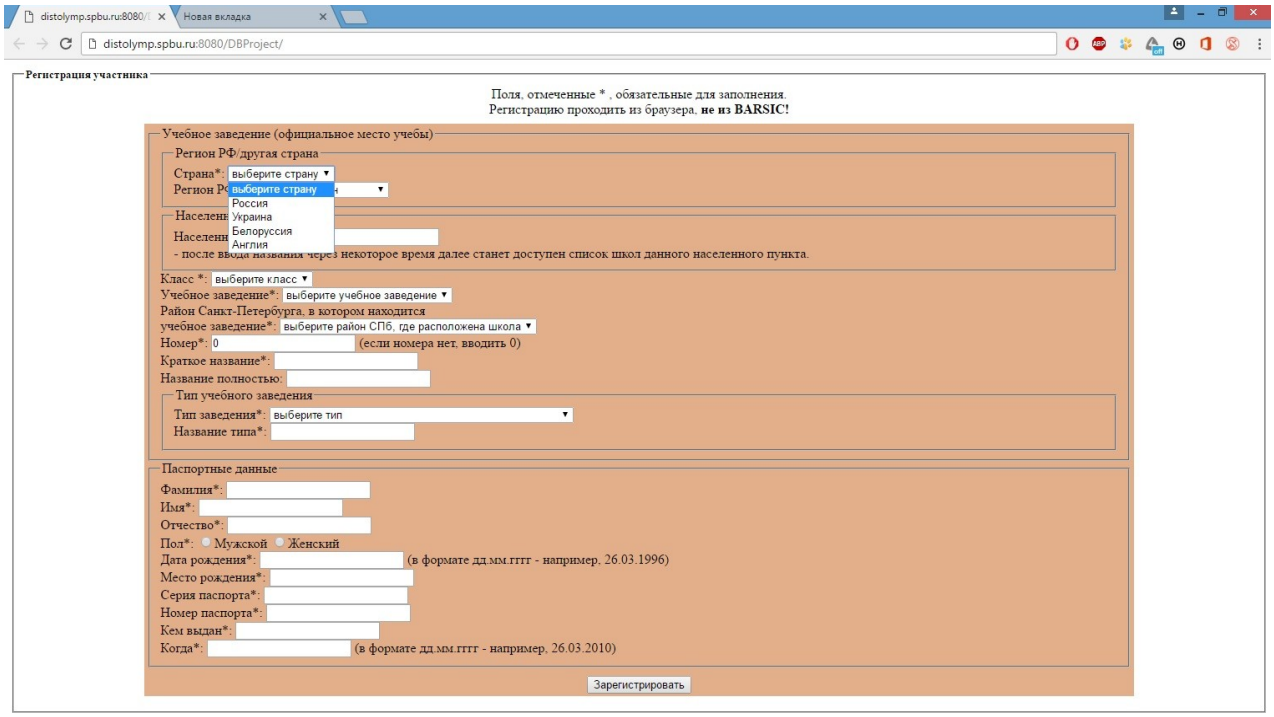

Рис.12. Работа реального проекта на сервере distolymp.

3.8 Обсуждение результатов

В результате работы удалось написать каркас будущей системы, к которому можно добавлять новые модули. Так как в работе использовались современные технологии (Spring, Maven, Hibernate) которые продолжают поддерживаться, расширение системы не должно вызывать затруднений. Общий механизм в каждом из слоев должен выглядеть следующим образом:

- слой View — написание новых jsp-файлов;
- слой Controller — создание системы контроллеров;
- слои DAO и Model — увеличение текущего количества классов и возможно создание новых методов, которые более полно решают задачи, возникающие в distolymp.

Структура проекта:

- в слое Model написано 14 классов, связанных с базой данных, и 2 класса для AJAX-запросов;
- в слое DAO написано 14 классов, 14 интерфейсов и 1 абстрактный класс;
- в слое Service написано 14 классов и 14 интерфейсов;
- в слое Controller написан один класс;

- в слое View написан один jsp-файл и 4 jsp-файла при работе над тестовым проектом;
- созданы 3 конфигурационных файла, файл pom.xml и два файла properties.

Наиболее важным направлением дальнейшей работы является решение ошибок, связанных с работой AJAX и обработкой исключений. Из более глобальных вопросов следует выделить проблемы безопасности и реализации работы с почтовыми сервисами. Эти проблемы не затрагивались в работе, но занимают важное место в работе PHP-версии distolymp. В Spring существуют механизмы решения проблем безопасности и работы с почтовыми сервисами — это платформа Spring Security и интерфейс MailSender [11].

Важно, что удалось добиться успеха в развертывании приложения на production-сервере. Таким образом, при дальнейшей разработке проекта появляется возможность проверки работоспособности приложения на реальном сервере без дополнительных проблем. Наличие файла, описывающего установку ПО на Debian, позволяет установить большую часть ПО на виртуальную машину для отладки.

Выводы по главе 3

- Были созданы все необходимые слои для работы приложения, использующегося для регистрации пользователей:

1. слой Controller, отвечающий за обработку запросов клиента;
2. слой DAO, отвечающий за взаимодействие с базой данных;
3. слой Service, отвечающий за бизнес-логику и связывающий Controller с DAO;
4. слой View, отвечающий за отображение результатов запроса клиенту;
5. конфигурационные файлы для работы приложения и корректной взаимосвязи всех слоев.

- Были решены следующие проблемы, возникшие по ходу разработки проекта:

1. отображение выпадающих списков в слое View для объекта model с атрибутами разного типа (проблема решена при помощи тега <spring:bind>);
2. проблема каскадного удаления записей в таблицах (проблема решена при помощи аннотации @OneToMany и корректирования методов в слое DAO);
3. невозможность создания таблицы passports из-за зарезервированных слов (проблема решена изменением названия столбцов).

- Были замечены следующие проблемы, которые не удалось решить:

1. аварийное завершение программы при вводе некорректных значений в поля формы (должна быть решена созданием обработчиков исключений на уровне контроллера, автор считает, что работу следует провести более полно, чем это было сделано при первичной разработке проекта, тем не менее, на уровне Model такие проблемы частично учтены при помощи аннотаций вида @Size);
2. некорректная работа технологии AJAX (проблема была описана в главе, посвященной View).

- Было установлено необходимое ПО на production-сервер: Apache Tomcat, Maven, JRE, JDK. Оно было проверено сначала на тестовом проекте, а затем на приложении для регистрации. Все возникшие ошибки были исправлены. По итогам работы на production-сервере был создан специальный файл с описанием команд для установки ПО и решения ошибок.

Выводы

- Проведен анализ технологий, применяемых для разработки Java-версий серверных программных комплексов, и выбран набор необходимых инструментов для создания Java-версии программного комплекса distolymp.

- Предложено использовать Apache Tomcat в качестве контейнера сервлетов и программные комплексы: Maven — для автоматизации сборки проекта, Spring — для слоя визуализации, Hibernate — для слоя обращения к базе данных, Mockito — для unit-тестирования.

- Написано приложение регистрации пользователей, которое будет каркасом для дальнейшего развития проекта.

- Помимо тестирования работы приложения на компьютере автора, проведена работа по развертыванию приложения на production-сервере. На production-сервере установлено необходимое для развертывания проекта ПО и проверена работа готового проекта. Все действия на production-сервере зафиксированы в специальном файле для удобства разработчиков, которые будут продолжать работу с проектом.

- Описаны дальнейшие цели в развитии проекта.

Литература

1. Монахов В.В., Максимов М.А., Саликов В.А., Зубов К.Р., Салатич А.А. DYSTOLYMP — программный комплекс для мультиплатформенной проверки знаний и практических умений // Современные информационные технологии. Теория и практика. Материалы II Всероссийской научно-практической конференции в рамках ИТ-форума «ICITY 2015: Информатизация промышленного города»., 2016. — с.124-129

2. Монахова Е. В. Усовершенствование системы генерации заданий и обработки результатов в программном комплексе distolymp. // Бакалаврская работа. СПбГУ, 2014 г., 23с.

3. Монахов В.В., Басов Л.В., Воропаев Р.А., Пивоваров А.М., Зуган М.С. Distolymp — программный комплекс для проведения интернет-олимпиад и дистанционного обучения. // В трудах XII Междунар. Конф. ФССО—2013, Петрозаводск, т.2, с.221-223.

4. Зуган М. С. Разработка инсталлятора и анализ безопасности программного кода комплекса distolymp. // Магистерская диссертация. СПбГУ, 2014, 4-16 с.

5. Басов Л. В. Разработка в программном комплексе distolymp подсистем назначения дипломов и регистрации пользователей. // Магистерская диссертация. СПбГУ, 2013, 9 с.

6. Э. Гонсалвес Изучаем Java EE 7. — СПб.:Питер, 2014. — с. 22-32

7. Блог Java программиста. Apache HTTPD, mod_jk, Apache Tomcat, Linux [Электронный ресурс] — Режим доступа: http://programador.ru/apache-mod_jk-tomcat/, свободный. —Яз. рус.

8. An overview of Tomcat 6 Servlet Container: Part 1 [Электронный ресурс] — Режим доступа: https://www.packtpub.com/books/content/overview-tomcat-6-servlet-container-part-1, свободный. —Яз.англ.

9. Apache Maven Project. О Maven [Электронный ресурс] — Режим доступа: https://www.apache-maven.ru, свободный. —Яз.рус.

10. Apache Maven Project. О Maven. Жизненный цикл сборки: фазы сборки [Электронный ресурс] — Режим доступа: http://www.apache-maven.ru/lifecycle.html, свободный. —Яз.рус.

11. К. Уоллс Spring в действии. —3-е издание. —М.: ДМК Пресс, 2013. — с. 32, 42, 433, 701

12. Spring Framework. Архитектура Spring [Электронный ресурс] — Режим доступа: https://spring-source.ru/docs_simple.php?type=manual&theme=docs_simple&docs_simple=chap01_p01, свободный. —Яз.рус.

13. JBoss documentation. Hibernate [Электронный ресурс] — Режим доступа: https://docs.jboss.org/hibernate/core/3.3/reference/en/html/objectstate.html, свободный. —Яз. англ.

14. Overview of Spring MVC Architecture [Электронный ресурс] — Режим доступа:

http://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html, свободный. — Яз. англ.

Приложение

В проекте разработано 74 класса, в качестве примеров далее приводится один из классов и фрагменты из двух других классов.

Примеры полей класса Schools.java из слоя Model:

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_school")
    protected int idSchool;

    @Column(name = "id_division", nullable = false)
    protected String idDivision;

    @Column(name = "id_town")
    protected String idTown;

    @Column(name = "id_district", nullable = false)
    protected String idDistrict;

    @Column(name = "number", nullable = false)
    protected String number="0";

    @Column(name = "id_type")
    protected String idType;

    @Column(name = "title", nullable = false)
    protected String title;

    @Column(name = "full_name", nullable = false)
    protected String fullName;

    @Column(name = "visible", nullable = false)
    protected String visible="yes";

    @Column(name = "editing", nullable = false)
    protected Boolean editing = true;

    @OneToMany(mappedBy = "idSchool", fetch = FetchType.LAZY, cascade =
CascadeType.ALL)
    protected Set<Groups> groups;

    @OneToMany(mappedBy = "idSchool", fetch = FetchType.LAZY, cascade =
CascadeType.ALL)
    protected Set<Users> users;

   @OneToMany(mappedBy = "idSchoolCurator", fetch = FetchType.LAZY, cascade =
CascadeType.ALL)
    protected Set<Informations> informations;

    @OneToMany(mappedBy = "idSchool", fetch = FetchType.LAZY, cascade =
CascadeType.ALL)
    protected Set<Registrators> registrators;
```

Класс TownsDaoImp.java:

```
package com.mycompany.dbproject.spring.dao;
```

```
import com.mycompany.dbproject.spring.model.Towns;
import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Query;
import org.hibernate.criterion.Restrictions;
import org.springframework.stereotype.Repository;

/**
 *
 * @author Admin
 */
@Repository("townDao")
public class TownsDaoImp extends AbstractDao <Integer, Towns> implements
TownsDao{

     public Towns findByIdTowns(int idTown){
        Criteria criteria = getSession().createCriteria(Towns.class);
        criteria.add(Restrictions.eq("idTown",idTown));
        return (Towns) criteria.uniqueResult();
    }

    public void saveTowns(Towns towns) {
        persist(towns);
    }

    public void deleteTowns(int idTown){
        Query queryDistrict = getSession().createSQLQuery("delete from districts
where id_town = :idTown");
        queryDistrict.setInteger("idTown", idTown);
        queryDistrict.executeUpdate();
         Query queryTown = getSession().createSQLQuery("delete from towns where
id_town = :idTown");
        queryTown.setInteger("idTown", idTown);
        queryTown.executeUpdate();

    }

    public List<Towns> getAllTowns(){
        Criteria criteria = createEntityCriteria();
        return (List<Towns>) criteria.list();
    }

    public Towns findByNameTowns(String town){
        Criteria criteria = getSession().createCriteria(Towns.class);
        criteria.add(Restrictions.eq("name",town));
        return (Towns) criteria.uniqueResult();
    }

//нереализованный метод из php-версии distolymp
    public void isValidData(){
        // если name пусто, то ошибку: $errors['division_name'] = 'Не заполнено
поле "Раздел"';
    }

}
```

Главные методы класса RegistrationController из слоя Controller:

```
    @RequestMapping(value = {"/registration","/"}, method = RequestMethod.GET)
    public String newUser(ModelMap model) {
        Users user = new Users();
        Countries country = new Countries();
        Schools school = new Schools();
```

```
        Passports passport = new Passports();
        List<Countries> listCountries = countriesService.getAllCountries();
        List<Regions> listRegions = regionsService.getAllRegions();
        List<Classes> listClasses = classesService.getAllClasses();
        String idTown = townsService.getIdTowns("Петергоф");
        List<Schools> listSchools = schoolsService.getAllSchoolsByTown(idTown);
        List<Schools> listSchools = schoolsService.getAllSchools();
       List<Districts> listDistricts = districtsService.getAllDistricts();
     List<SchoolTypes> listSchoolTypes = schoolTypesService.getAllSchoolTypes();
        model.addAttribute("user", user);
        model.addAttribute("country", country);
        model.addAttribute("school", school);
        model.addAttribute("passport", passport);
        model.addAttribute("listCountries", listCountries);
        model.addAttribute("listRegions", listRegions);
        model.addAttribute("listClasses", listClasses);
        model.addAttribute("listSchools", listSchools);
        model.addAttribute("listDdistricts", listDdistricts);
        model.addAttribute("listSchoolTypes", listSchoolTypes);
        model.addAttribute("edit", false);
        return "reg";

    }

    @RequestMapping(value = {"/registration","/"}, method = RequestMethod.POST)
     public String saveUser(Users user, Passports passport, Countries country,
Schools school, BindingResult result,
            ModelMap model) {
        if (result.hasErrors()) {
            return "reg";
        }
        if(country.getName()!= null){
           countriesService.saveCountries(country);
        }
        LocalDateTime dateReg = new LocalDateTime();
        school.setIdDivision("1");
        school.setIdTown("1");
        passport.setCode("zaglushka");
        passportsService.savePassports(passport);
        if(passport.getFirstName()!= null){
        user.setIdPassport(passportsService.getLastIdPassport());
        user.setIdDivision("1");
        user.setIdContact("1");
        user.setIdInfo("1");
        user.setIdPassport("1");
        user.setIdReg("1");
        user.setRegIp("1");
        user.setRegDate(dateReg);
        user.setMainpass("1");
        user.setLogin("username " + passport.getN1());
        user.setPassword("1111" + passport.getN1());
        usersService.saveUsers(user);
         model.addAttribute("success", "User " + user.getLogin() + " registered
successfully");
        }
        return "reg";

    }
```