Содержание

0. Введение

1. Создание пустого проекта

2. Добавление тестового контроллера

3. Добавление необходимых моделей

4. Добавление необходимых интерфейсов сервисов

5. Добавление реализаций сервисов на HashMap

6. Добавление тестовых данных

7. Добавление ContestController

8. Добавление индексной страницы

9. Добавление списка контестов

10. Добавление страницы списка задач контеста

11. Добавление страницы списка посылок по контесту

12. Перенос всех данных в базу данных

13. Переписывание сервисов на работу с репозиториями БД

14. Добавление авторизации

15. Добавление плашки с активным пользователем

16. Добавление возможности отправки решения

17. Добавление регистрации

18. Добавление проверки на валидность username/password

19. Добавление шифрования пароля

20. Добавление возможности просмотра кода

21. Итоги

Введение

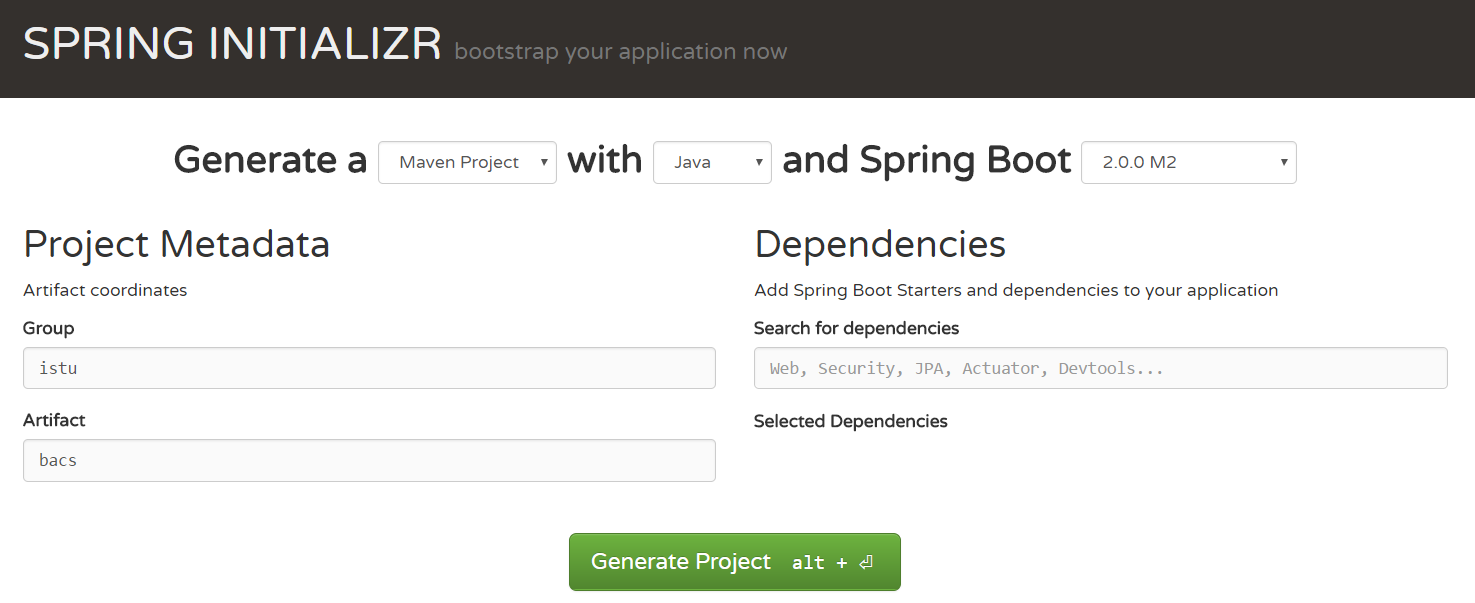
В данном руководстве будет рассмотрен процесс создания веб-приложения с нуля. Стек используемых технологий – Java 8 + Spring.

В процессе данного руководства будет разрабатываться облегчённая версия тестирующей системы BACS (<http://bacs.cs.istu.ru/>).

Весь код, использованный в руководстве, можно найти по адресу <https://github.com/PloadyFree/bacs-tutorial>. История коммитов разработана таким образом, что в сообщении каждого коммита есть его номер. Этот номер совпадает с номером части, в которой делаются те или иные изменения. Рекомендуется после «прохождения» каждой части сверяться с тем, что есть в репозитории.

Создание пустого проекта

Начнём с того, что необходимо создать сам проект, в котором будем работать. Для этого зайдёт на <http://start.spring.io/> и создадим пустой проект, как показано на рисунке ниже:



Скачается архив с единственной папкой ‘bacs’, распакуем её, например, в ‘C:/my/bacs’.

На данном этапе в проекте содержится два файла с именем mvnw – это файлы менеджера зависимостей maven, которым мы будем пользоваться в будущем.

Там же лежит `pom.xml’, в котором прописаны некоторые из используемых зависимостей (например, `spring-boot-starter-test`), а также некоторые свойства проекта, такие как его имя, описание, версия и др.

В самом конце указаны репозитории, к которым в будущем будет обращаться maven для того, чтобы найти и скачать нудные нам зависимости.

Добавление тестового контроллера

Для того, чтобы наш сервис мог отдавать браузеру информацию, нам потребуются контроллеры. Контроллер – это класс, описывающий поведение сервиса при каком-либо запросе к нему.

Во-первых, переименуем `application.properties` в `application.yml`. Yaml-формат является более удобной альтернативой формата `.properties`.

В application.yml установим порт, на котором будет работать сервис. Заполним его следующим содержимым:



Затем создадим папку web в `/src/main/java/istu/bacs`, а внутрь неё положим следующий класс контроллера:



Контроллеры обычно называются со словом Controller на конце.

Здесь мы видим аннотацию @RestController над классом. Она говорит о том, что данный класс является контроллером, а все его методы, помеченные через @RequestMapping возвращают некоторый «сырой результат”, то есть текст, либо сериализованную в json сущность. Сам @RequestMapping(“/”) говорит о том, что этот метод будет отвечать на запрос к корню сервиса и возвращать строку “Hello, Spring!”.

Теперь, когда мы запустим этот сервис (рекомендуется запускать его через IntelliJ IDEA), в браузере зайдём по адресу <http://localhost/>, увидим сообщение “Hello, Spring!”.

Добавление необходимых моделей

Для того, чтобы как-то организовать данные, которыми мы будем пользоваться, создадим некоторую структуру сущностей.

Для этого создадим в `src/main/java/istu/bacs` директорию `model`. Внутри неё будут располагаться сущности.

Также предварительно добавим зависимость lombok в application.yml. Для этого в application.yml в секции dependencies добавим следующее:



Эта зависимость поможет избавиться от лишнего кода (бойлерплейта) в сущностях.

Первым классом будет класс пользователя User.



Здесь аннотации @Getter, @Setter и @AllArgsConstructor добавляют этому классу геттеры и сеттеры ко всем полям, а также конструктор, принимающий userId и username.

Теперь добавим класс задачи Problem:



Следующим будет класс контеста Contest:



Далее нам потребуются дополнительные типы, а именно классы вердикта и языка.

Создадим директорию `src/main/java/istu/bacs/model/type` и положим в неё эти два класса.

Класс языка Language:



Класс вердикта Verdict:



Теперь добавим оставшийся класс посылки Submission в `model`:



Добавление необходимых интерфейсов сервисов

Теперь, для того, чтобы управлять сущностями, созданными в предыдущем разделе, необходимо создать сервисы. Начать следует с определения их интерфейсов, чтобы определить, как сущности будут взаимодействовать с проектом в целом.

Создадим директорию `src/main/java/istu/bacs/service`, в ней будут лежать наши сервисы.

Первым будет сервис управления пользователями:



Этот сервис умеет искать пользователя по его id, а также сохранять нового пользователя. Этого будет достаточно на текущий момент.

Кроме того, в самом начале каждый из последующих сервисов будет иметь только эти два метода.

Теперь создадим сервис управления задачами:



Сервисы управления задачами и контестами ничем различаться с двумя выше показанными сервисами не будут.

Добавление реализаций сервисов на HashMap

Для того, чтобы воспользоваться сервисами, определёнными в предыдущей части, необходимо написать их реализации. Наши реализации будут достаточно простыми, основанными на хранении данных в HashMap.

Первым делом необходимо создать директорию `src/main/java/istu/bacs/service/impl`. В ней разместим наши реализации.

Здесь будут описано определение только сервиса управления пользователями. Сервисы для управления задачами, посылками и контестами будут выглядеть аналогично.

Реализация сервиса управления пользователями:



Здесь у нас имеется словарь Map<Integer, User> userById, он хранит пользователей по их userId.

Аннотация @Service говорит спрингу о том, что этот класс является сервисом. Таким образом, когда нам в каком-нибудь месте понадобится сервис UserService, нам не надо будет указывать конкретную реализацию (UserServiceImpl), спринг сам создаст экземпляр этого класса. Такой подход удобен тем, что, если мы захотим заменить реализацию UserServiceImpl на что-то другое, не придётся изменять код во всех местах использования этого сервиса, нужно будет лишь написать новый сервис и он уже будет работать.

Добавление тестовых данных

Теперь неплохо бы добавить тестовые данные. Сделаем это, создав следующий класс в `src/main/java/istu/bacs/` (напомню, что этот класс можно найти в репозитории на GitHub):









Этот класс помечен аннотацией @Component, которая говорит спрингу, что этот класс нужно проверить и выполнить какие-то действия. Также он реализует интерфейс ApplicationRunner, в котором определён метод run(), этот метод будет запущен спрингом при инициализации, то есть перед началом работы сервиса. Этим мы добились того, что данные будут доступны сразу после запуска приложения.

Добавление ContestController

На данный момент мы имеем все необходимые сущности, сервисы, позволяющие обращаться с этими сущностями, но мы никак не может достучаться до них через веб-интерфейс. Добавим небольшой контроллер, роль которого будет заключаться в том, чтобы вывести наши тестовые данные. Он будет лежать в папке `web`:



Этот контроллер умеет отвечать на запрос `/contests`, `/contest/123` и `contest/123/submissions`.

На первый запрос он будет отдавать список контестов в базе, на второй список задач для указанного контеста, а на третий все посылки по указанному контесту.

В методе getSubmissionsForContest мы видим {contestId} в пути и @PathVariable перед аргументом Integer contestId. Фигурные скобки позволяют нам определять переменные, которые буду находиться в пути, в данном случае мы определили, что на месте {contestId} будет стоять номер контеста. Само же значение номера контеста будет находиться в аргументе contestId.

Заметим, что мы пользуемся методами сервисов, которых у нас ещё нет. Добавим их. В ContestService добавим метод ` List<Contest> findAll();`, а в его реализацию ContestServiceImpl добавим его реализацию:



Также у нас используется метод ` SubmissionService.findAllByContestId`, который также ещё не был нигде определён. Добавим метод ` List<Submission> findAllByContestId(Integer contestId); ` в интерфейс SubmissionService, а также его реализацию в Impl класс:

Теперь мы можем запустить наше приложение и направиться, например, на <http://localhost/contests> или на <http://localhost/contest/1/submissions>. В местах, где должно быть время, мы можем увидеть большой объект с кучей непонятных полей. Нам это сейчас не важно, в будущем мы научимся это обрабатывать.

Добавление индексной страницы

Согласитесь, не очень удобно получать информацию в том виде, в котором мы получали её в предыдущих частях. В этой части мы научимся создавать html-страницы и пользоваться ими в рамках приложения.

Для начала добавим две зависимости в наш pom.xml:



Первая зависимость позволяет использовать шаблонизаторы (шаблонизатор – это способ совмещать некоторый подготовленный шаблон html-страницы с данными, находящимися на сервере). Вторая зависимость позволит включать одни шаблоны в другие, что тоже очень удобно.

Проведём небольшие изменения в нашем контроллере MainController.

Заменим в нём @RestController на @Controller и будем возвращать из метода home() строку “index”. Index – это основная страница приложения.

Теперь в папке `src/main/resources/templates` создадим наши страницы.

Первая будет даже не самостоятельной страницей, а «шапкой», которая будет включаться во все страницы. Разместим на ней навигацию. Создадим файл `default-header.html` со следующим содержимым:



В теге html мы определили два пространства имён, которые будут нами использоваться.

layout – он нужен для того, чтобы определять в файле те фрагменты, которые в будущем можно будет использовать из других файлов. Так, например, мы с его помощью в теге nav определили, что эта часть страницы будет доступна нам по имени `default-header`.

th – он нужен для использования самого шаблонизатора. С его помощью мы можем создавать ссылки на наши методы, создавать таблицы и списки и много другое (со всеми возможностями шаблонизатора можно ознакомиться на <http://www.thymeleaf.org/>).

Следующим будет файл main-layout.html:



Здесь мы определяем в head.title способ описания имени вкладки. Шаблон "$CONTENT\_TITLE - $LAYOUT\_TITLE" говорит, что сначала будет применено имя основной страницы, потом будет тире, и затем будет имя шаблона (BACS).

layout:replace="~{default-header :: default-header}" означает, что тег, содержащий эту строку, будет заменен частью, определённой на странице default-header, с названием default-header.

layout:fragment="content" означает, что если страница будет использовать текущую страницу как шаблон, то она будет заменять часть страницы, помеченной “content”.

Теперь определим саму страницу index.html:



Здесь layout:decorate=”~{main-layout}” означает, что данная страница будет декорирована страницей main-layout, а layout:fragment=”content” означает, что эта часть кода будет заменять фрагмент “content” у применённого шаблона (т.е. у main-layout.html).

Теперь, если мы запустим сервис и зайдём на <http://localhost/>, то увидим отрисованную шаблонизатором страницу:



Добавление списка контестов

Для того, чтобы добавить страницу контестов, нам потребуется не так уж много. Однако, для того, чтобы в связке с Thymeleaf форматировать даты так, как нам нужно, потребуется добавить ещё одну зависимость, расширяющую возможности работы с датами:



Пускай страницы, отвечающие за контесты, будут лежать в `src/main/resources/templates/contests`.

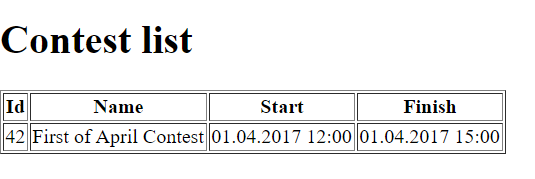
Добавим туда страницу `contest-list.html`:



Задекорируем нашу страницу основным main-layout’ом, а сами контесты будут располагаться в таблице с колонками id, имя контеста, время начала и время окончания.

Таблица строится с помощью th:each, который итерируется по коллекции contests (которую мы чуть позже будем передавать в модель), а с помощью ${name} можно обращаться к различным переменным из модели. #temporals – это и есть расширение, позволяющее работать с датами. В данном случае мы описываем, как форматировать дату и передаём саму дату в метод format. Кроме того, есть возможность задавать текст в тегах, который будет использоваться по умолчанию, то есть эту страницу можно открывать в браузере без запуска приложения, и она будет отображать те данные, которые записаны в файле страница. При обработке этой страницы Thymeleaf’ом, он заменит текст данными из модели.

Без обработки Thymeleaf’ом страница выглядит так («шапки» здесь нет, так как её вставляет шаблонизатор):



Также добавим ссылку на нашу страницу в файл default-header, чтобы возможность просмотра списка контестов была доступна из навигационной панели. В default-header.html после ссылки HOME добавим следующую строку:



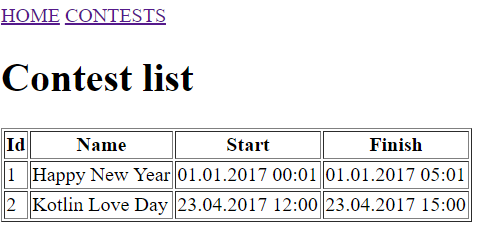
Теперь осталось добавить обработку запроса /contests в наш контроллер. Для этого заменим содержимое ContestController на следующее:



Здесь константа VIEWS\_CONTEST\_LIST указывает, где находится страница со списком контестов.

Метод getAllContests(Model model) обрабатывает на запрос /contests. Также он принимает объект модели model, в которой можно хранить все объекты, с которыми мы собираемся работать при отрисовке страницы. Затем возвращается адрес страницы, Thymeleaf отрисовывает страницу и отдаёт пользователю.

Если мы зайдем на <http://localhost/contests>, страница будет выглядеть следующим образом:



Добавление страницы списка задач контеста

Теперь было бы неплохо иметь возможность увидеть список задач контеста. У нас будет простая страница с этими задачами, для каждой из которых будет показана основная информация по задаче, такая как номер в списке задач контеста, название и другие.

Так как мы начали делать наши страницы на шаблонах, сделаем шаблоны для контестов.

Все страницы, напрямую связанные с контестами, будут лежать в `src/main /resources/templates/contests/`.

Начнём с навигационной панели контеста contest-header.html:



В нём указываем, что этот элемент nav будет доступен через имя `contest-header`. Пока что будет ссылка только на список контестов. Укажем через href=”contest-problems.html” ссылку на статичную страницу (которую мы добавим чуть позже), которую можно будет открыть в браузере, не запуская сам сервер, а шаблонизатору скажем, что эта ссылка будет вести по пути /contest/contestId. Таким образом, мы можем нажать на ссылку CONTEST в браузере и попадём напрямую на страницу с задачами контеста, но, когда будет запущен сервер, страницу от сервера, браузер по нажатию ссылки перенаправит запрос в метод контроллера, обрабатывающий этот запрос (/contest/contestId). На данном этапе контроллер ещё не умеет обрабатывать запросы такого типа, но мы научим его это делать в конце этой части.

Теперь опишем contest-layout.html, которым будут декорироваться все страницы, относящиеся к контесту:



Этот шаблон является вложенным, то есть он сам декорирует main-layout, но, в то же время, будет являться декоратором для других страниц. Так как Thymeleaf не умеет обрабатывать имя вкладки для вложенных шаблонов, определим описание имени вкладки ещё раз (на самом деле, оно ничем не отличается от того, что было в main-layout). Указываем часть, которая будет декорировать content из родительского шаблона, а внутри неё расположим сначала навигационную панель для контеста, а затем контент контеста, в котором и будет располагаться список задач.

Теперь добавим саму страницу списка задач:



Также добавим возможность нажать на имя контеста в списке контестов и попасть на список задач.

Заменим в contest-list.html строку 26 (с именем контеста) на следующее:



Единственное, что осталось сделать – написать обработку запроса списка контестов.

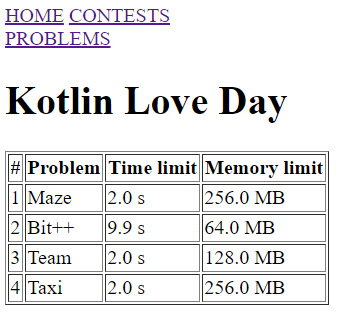
Добавим с ContestController сразу же после строки VIEWS\_CONTEST\_LIST путь к странице со списком контестов:



А также добавим сам метод, обрабатывающий запрос:



Теперь можем зайти на <http://localhost/contests>, и кликнуть по одному из контестов:



Добавление страницы списка посылок по контесту

Теперь, когда у нас есть список задач по контесту, научимся отображать список посылок, относящихся к конкретному контесту.

Так как на предыдущем шаге мы сделали необходимые шаблоны, для всего, что относится к контесту, теперь, чтобы добавить список посылок, нам придётся приложить минимум усилий.

Все страницы, относящиеся к посылкам, будут лежать в `src/main/resources/templates/contests/submissions`.

Добавим страницу с посылками submission-list.html:



Затем добавим в contest-header.html ссылку на список посылок. После этого contest-header.html станет выглядеть следующим образом:

Также заметим, что изнутри html-страницы можно обращаться не только к тому, что мы положили в модель, но и, например, к отдельным частям запроса, в данном случае – к contestId.

Теперь займёмся контроллером ContestController.

Добавим путь к только что созданной странице:



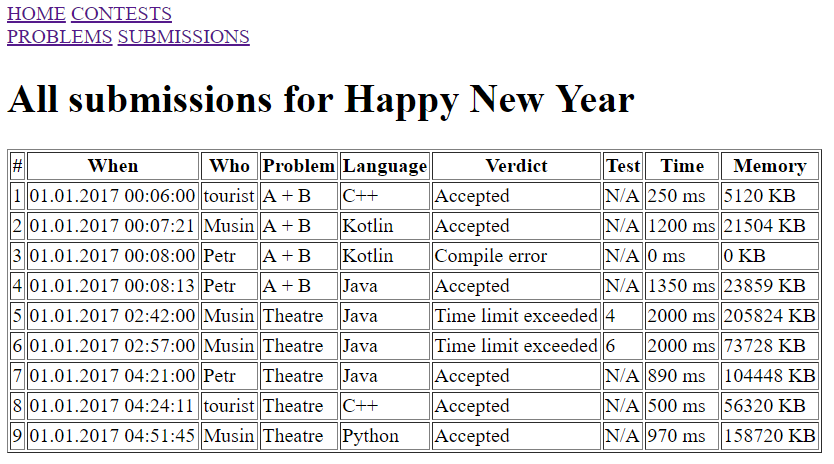
Также нам понадобится SubmissionService, так что изменим наш контроллер следующим образом:



Осталось добавить сам обработчик:



Для проверки результата запустим сервер и зайдём на <http://localhost/contest/1/submissions>:



Перенос всех данных в базу данных

Сейчас наше приложение уже работает и мы можем получать из него некоторую информацию. Однако, эта информация создаётся при старте сервера и теряется при его остановке. Для того, чтобы всегда иметь все необходимые данные, нам надо их хранить отдельно от приложения.

Для того, чтобы достичь этой цели, будем использовать базу данных. Напишем два скрипта: первый для того, чтобы создать нужные таблицы, и второй, чтобы заполнить их данными.

Сами скрипты будут находиться в `src/main/resources/`.

Скрипт для создания таблиц schema.sql:



Скрипт для заполнения таблиц данными data.sql:







Опять же, эти скрипты лучше взять с репозитория на GitHub.

В следующих частях мы научимся использовать эти скрипты.

Переписывание сервисов на работу с репозиториями БД

В этой части мы перенесём все наши данные в базу данных.

Для начала требуется установить mariadb версии не ниже 10 с оф. сайта <https://mariadb.org/>.

Также надо будет создать базу данных с именем `bacs`.

Здесь и далее будет подразумеваться, что в базе существует пользователь с именем `root` и паролем `root, а также, что БД работает на порту 3306 (он будет установлен по умолчанию). Все эти данные можно установить при первоначальной настройке mariadb.

Приступим.

Так как у нас уже есть скрипты, заносящие данные в базу, удалим класс DataLoader за ненадобностью.

Пропишем в application.yml настройки подключения к базе:



Здесь свойство continue-on-error: true нужно для того, чтобы скрипт, добавляющий данные в базу, смог работать при втором и следующих запусках. Без этого свойства будет происходить ошибка, говорящая, что запись с указанным id уже существует и сервер не запустится. Свойство user-new-id-generator-mappings: false нужно для того, чтобы id новых записей, для которых не указан id, создавался корректно.

Добавим в pom.xml две зависимости. Первая позволяет использовать Hibernate (библиотеку для работы с базой данных из кода), вторая позволяет общаться с базой mysql напрямую:



Чтобы воспользоваться возможностями Hibernate, создадим интерфейс репозитория SubmissionRepository (все репозитории находятся в `src/main/java/istu/bacs/repository`:



Этот интерфейс наследуется от JpaRepository<Submission, Integer>. Submission – это сущность, с которой этот репозиторий будет работать, а Integer – это тип поля id этой сущности (у Submission есть Integer submissionId).

В этом интерфейсе мы определили метод findAllByContest(Contest contest). Этот метод будет искать все посылки, относящиеся к указанному контесту. Самое интересное – это то, что для того, чтобы воспользоваться этим методом, больше ничего писать не нужно, Hibernate сделает за нас.

В самом же интерфейсе JpaRepository определено множество методов, позволяющих найти сущность по id, найти сущности по какому-то признаку, удалить сущности и многое другое.

Таким же образом создадим остальные репозитории.

ContestRepository:



ProblemRepository:



UserRepository:



Наши сервисы сейчас работают на HashMap. Для того, чтобы они работали напрямую с базой данных, перепишем их.

SubmissionServiceImpl:



Этот сервис теперь принимает в конструкторе SubmissionRepository, который для нас создаст Hibernate при инициализации приложения. Все методы перенаправляют обязанности по поиску и изменению информации этому репозиторию.

Оставшиеся репозитории переписываются похожим образом.

Теперь переделаем сущности для того, чтобы связать их с конкретными таблицами в базе данных.

Начнём с пользователя User:



Сущности были добавлены несколько аннотаций.

@NoArgsConstructor говорит о том, что теперь у сущности существует конструктор без параметров. Этого требует Hibernate.

@Entity говорит о том, эта сущность связана с таблицей `user` в базе данных.

@Id и @GeneratedValue нужны для определения поля, являющегося первичным ключом в базе. GenerationType.AUTO указывает, что способ генерации ключа задаёт сама база.

Теперь рассмотрим класс Problem:



В нём произошли ровно те же изменения, что и в User.

С классом Submission дела обстоят несколько сложнее:



Здесь была добавлена аннотация @ManyToOne. Если рассматривать её на поле author, то это значит, что у одного пользователя может быть много посылок, но любая конкретная посылка принадлежит только одному пользователю.

@JoinColumn(name = “author\_id) говорит, что в таблице с посылками существует столбец author\_id. Когда мы запросим посылку, Hibernate посмотрит на это поле и запросит из базы пользователя, у которого user\_id = author\_id, то есть «привяжет» пользователя к посылке. Тогда, даже несмотря на то, что посылки и пользователи хранятся в разных таблицах, мы можем не обращать на это внимания и работать, будто они определены в одном месте.

Также стоит заметить, что Hibernate сам не в состоянии корректно обрабатывать поля типа Verdict и Language, поэтому напишем для них конвертеры и положим их в `src/main/java/istu/bacs/model/converter`.

VerdictConverter:



LanguageConverter:



Это классы автоматически подхватятся спрингом при инициализации.

@Converter(autoApply = true) говорит о том, что данный класс является конвертером. Любой конвертер должен реализовывать интерфейс AttributeConverter<Type, DbType>, тем самым определяя методы, с помощью которых будет производиться конвертация значений.

Ещё немного нового функционала используется в сущности Contest:



Во-первых, здесь применяется аннотация @ManyToMany, которая говорит, что одна задача может быть в любом количестве контестов, и наоборот, любой контест может иметь множество задач.

Во, вторых, здесь применяется @JoinTable – механизм объединения таблиц. Если мы посмотрим в schema.sql, то увидим, что там есть таблица `contest\_problems`:



При связях многие-ко-многим, эта таблица позволяет определить эти самые связи. В этой таблице 3 столбца: contest\_id для определения контеста, к которому привязывается задача, problem\_id для определения, какая именно задача привязывается, и order для определения номера задачи в контесте.

Аннотация @JoinTable принимает имя таблицы-связки в name, а также joinColumns и inverseJoinColumns, которые определяют имена колонок, с помощью которых производится свзяка.

@OrderColumn, в свою очередь, позволяет указать, на основе какого столбца определяется порядок элементов.

На этом перенос данных в базу завершён.

Также хочу заметить, что Hibernate автоматически запустит скрипты schema.sql и data.sql при инициализации приложения, так что эти имена скриптов были выбраны не случайно.

Добавление авторизации

Любой современный веб-сервис имеет возможность авторизации. Добавим её и здесь.

Для этого нам понадобится зависимость Spring Security:



Сначала добавим нашим пользователям пароль. Для простоты он будет пустым.

Внесём изменения в схему БД schema.sql, добавив столбец с паролем каждому пользователю:



А также сам пароль в data.sql:



Для того, чтобы воспользоваться Spring Security, сделаем так, чтобы User реализовывал интерфейс UserDetails, который позволяет хранить полезную информацию о статусе учётной записи пользователя:



Также необходимо сделать так, чтобы UserService реализовывал UserDetailsService:



В его реализации (UserServiceImpl) добавится следующий метод:



Он будет возвращать пользователя по username, но только в случае, если такой существует. Иначе он будет выбрасывать UsernameNotFoundException.

Здесь мы воспользовались методом, которого ещё не было в UserRepository, поэтому добавим его туда:



Опять же, Hibernate достаточно умён, чтобы сгенерировать корректный запрос к базе при вызове метода findByUsername.

Осталось добавить только конфигурацию Spring Security. Положим её в `src/main/java/istu/bacs/security`:



Разберём эту конфигурацию.

В методе configure(HttpSecurity http) описаны правила безопасности для http-соединений.

`formLogin()` говорит о том, что для авторизации нужно использовать страницу авторизации. Так как мы не указываем, какую именно страницу авторизации мы хотим, Spring Security сгенерирует её сам.

`authorizeRequests().antMatchers(“/”).permitAll()` говорит о том, что нужно разрешить доступ к корню сайта любому пользователю.

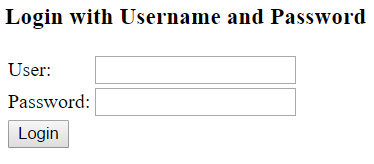
`authorizeRequests().anyRequest().hasAuthority(“ROLE\_USER”)` говорит о том, что доступ по любому другому пути разрешён только пользователям, имеющим права “ROLE\_USER”.

Теперь можем удалить таблицу с пользователями из нашей базы, чтобы она создалась заново. Это нужно, чтобы добавился столбец с паролем.

На этом конфигурирование Spring Security закончено, можно проверить, заработало ли всё. Для этого попробуем зайти на <http://localhost/>:

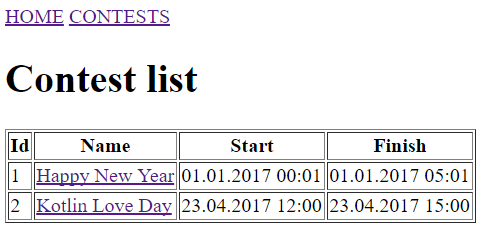


Однако, при попытке зайти, например, на страницу со списком контестов, кликнув на CONTESTS в верней панели, получим следующее:



Это и есть страница, сгенерированная Spring Security.

Если мы введём, например, User: tourist, пароль оставим пустым, то мы получим доступ ко всему остальному содержимому сайта:



Добавление плашки с активным пользователем

В предыдущей части мы научились авторизовываться в нашем приложении. Однако, у нас пока что нет возможности сменить пользователя или хотя бы видеть, какой пользователь сейчас авторизован.

Для того, чтобы отображать информацию об авторизации, нам понадобится зависимость:



Добавим в templates html-страницу active-user.html, которая и будет отображать текущего пользователя:



Здесь sec:authorize=”authenticated” означает, что этот элемент будет отображаться, только если пользователь авторизован. sec:authentication="principal.authorities" означает, что данный элемент будет заменен списком прав пользователя.

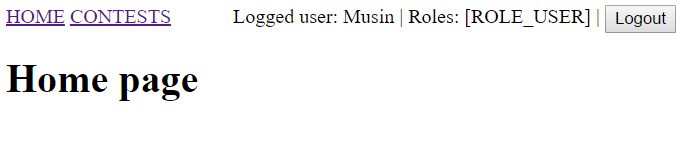
Теперь добавим эту плашку в default-header, чтобы она отображалась всегда:



Теперь можем запустить сервер и зайти на <http://localhost/>:



И, если мы войдём в систему, то увидим примерно следующее:



Добавление возможности отправки решения

Теперь, когда мы научились входить в систему, научимся пользоваться этим.

В этой части появится возможность отправить решение на какую-либо задачу.

Начнём с контроллера. Добавим в ContestController строку, описывающую положение страницы, отвечающей за отправку решения:



Также добавим метод, возвращающий форму отправки решения:



Этот метод будет отлавливать запросы по адресу `/contest/123/submit`, где 123 – номер контеста. Он кладёт в модель пустую посылку, которую нам надо будет заполнить на странице, а также список языков и задач. Можно заметить, что этот метод помечен как @GetMapping. Это значит, что данный метод будет обрабатывать только GET-запросы, то есть когда нас просят вернуть страницу. Когда же нас просят принять данные, запрос будет иметь POST-тип. Именно таким и является метод, обрабатывающий посылку:



Этот метод принимает MultipartFile, который позволит загружать код решения из файла. @RequestParam говорит, что клиент(браузер) готов отправлять файл. @AuthenticationPrincipal User user позволяет получить авторизованного пользователя.

В теле метода посылка заполняется тестовыми данными и сохраняется в репозитории. В реальной системе должен быть ещё сервис, который проверяет посылку и обновляет её статус, использованное время и память и т.п., но мы этим, для просты, заниматься не будем.

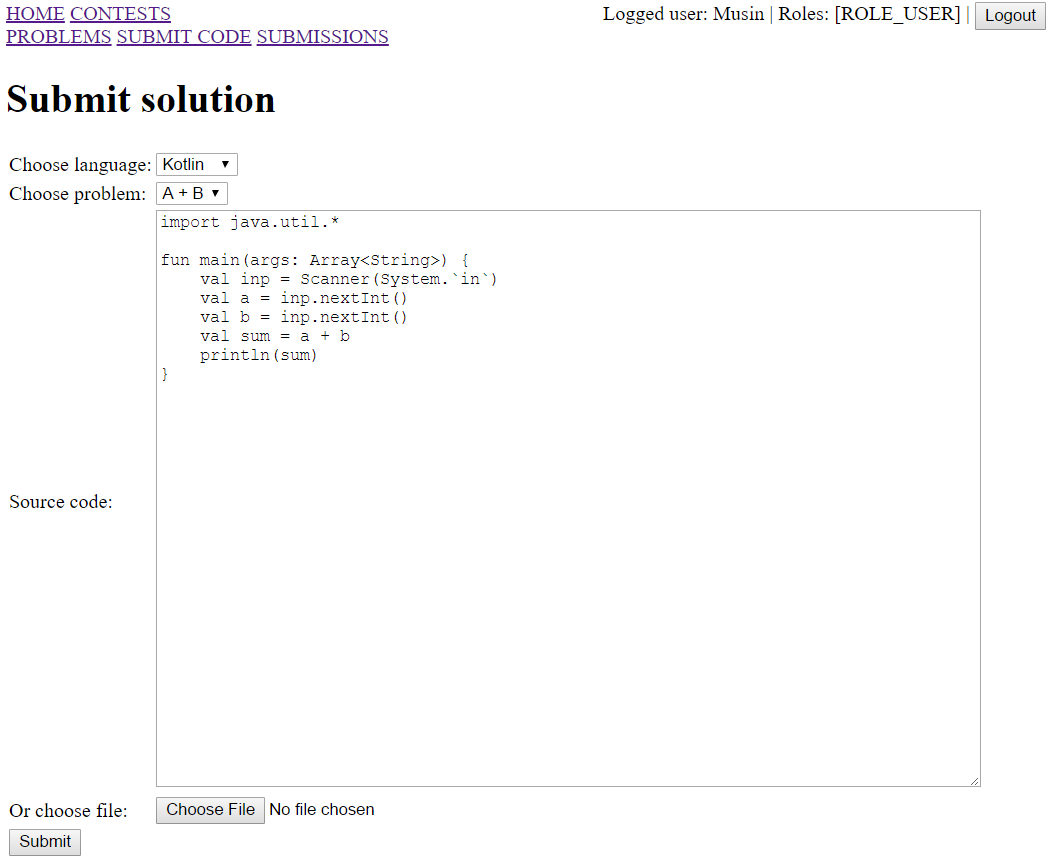
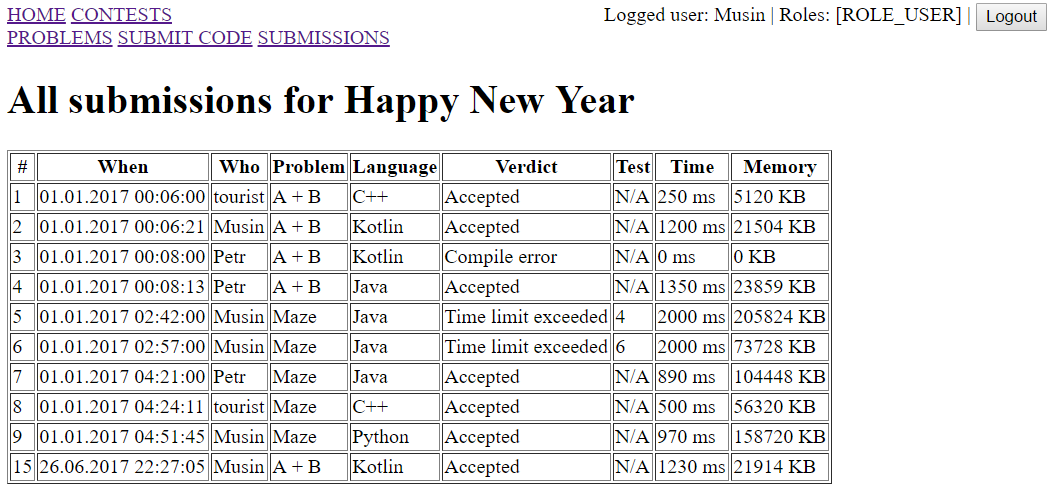
Метод возвращает redirect на другой метод. Это значит, что дальнейшие действия будут приниматься методом, обрабатывающим запрос "/contest/{contestId}/submissions".

Теперь в `templates/contests/submissions` добавим страницу submit-solution:



Для возможности отправки файла необходимо в теге form также указать enctype="multipart/form-data".

Теперь можно запустить сервер, зайти на страницу контеста и отправить решение:

Добавление регистрации

Три пользователя – это хорошо, но на любом популярном веб-сервисе пользователей гораздо больше. Чтобы не добавлять пользователей в базу самостоятельно, добавим возможность регистрации.

Положим в templates страницу register.html:



При нажатии на кнопку Register произойдёт вызов метода /register.

Добавим точно так же страницу входа, чтобы иметь возможность пользоваться шаблонизатором. Страницу login.html положим в templates:



В сервисе UserService есть метод save(). Переименуем его в register(). В больших проектах так делать не надо, так как должна быть возможность изменять данные пользователя, но, для простоты, предположим, что данные пользователя не изменяются:



В реализации напишем следующее:



Также добавим в папку с сервисами service исключение UsernameAlreadyInUseException, которое будет говорить о том, что пользователь с данным username уже зарегистрирован:



В конфигурации SecurityConfig метод configure(HttpSecurity http) дополним следующим образом:



В нём мы разрешили доступ к методам /login и /register без авторизации, а также попросили Spring Security использовать /login для выдачи страницы авторизации.

Теперь изменим в active-user.html часть страницы, показывающуюся неавторизованному пользователю так, чтобы пользователь видел кнопку регистрации:



Единственное, что осталось сделать – добавить саму страницу регистрации. Положим её в templates:

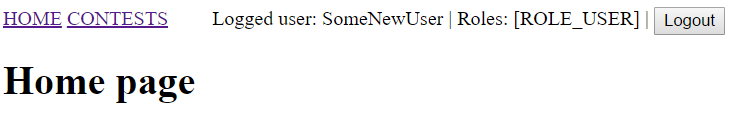


Вот и всё, возможность регистрации готова, испробуем на деле!

Запустим сервер и зайдём на <http://localhost/register>:



После успешной регистрации нас отправит на страницу с авторизацией, откуда мы сможем авторизоваться и получить доступ к сервису:



Добавление проверки на валидность username/password

В предыдущей части были созданы страницы регистрации и авторизации. Однако, если на странице авторизации ввести некорректные данные, то не будет понятно, что же пошло не так. В этой части мы добавим несколько проверок на валидность данных, в том числе и при регистрации. Для этого поставим ограничение при регистрации на длину логина и пароля от 6 до 20 символов, а также.

Начнём с обработки некорректной авторизации. В login.html добавим после открытия формы сообщение об ошибке, которое будет видно только если действительно произошли какие-либо ошибки:



В целях безопасности лучше никогда не сообщать пользователю, что именно пошло не так.

Так как Spring Security отправит нас на /login?error при неудачной авторизации, для того, чтобы проверить, есть ли ошибки, достаточно посмотреть, есть ли в параметрах запроса error.

Теперь добавим валидацию логина и пароля при регистрации.

Для этого в register.xml заменим тег table следующим:



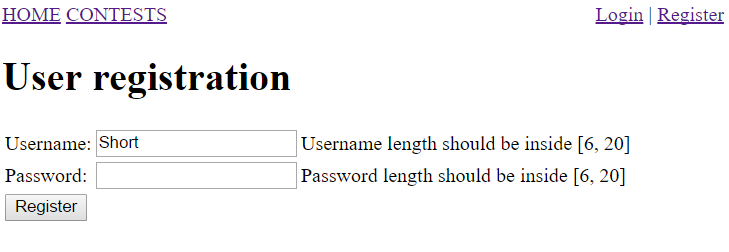
Теперь, если существуют ошибки, связанные с логином или с паролем, то они будут видны на странице.

Добавим теперь проверку данных при авторизации в AuthController, заменив метод register следующим:

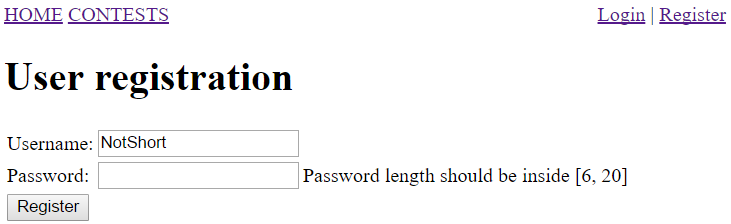


Здесь `errors.rejectValue’ принимает имя поля, имя ошибки и описание ошибки, которое и будет показано на странице.

Теперь можно запустить сервер и попробовать зарегистрировать пользователя со слишком коротким логином, получится примерно следующее:



И даже если логин будет удовлетворять требованиям, а пароль нет, то всё равно зарегистрироваться не удастся:



Добавление шифрования пароля

Сейчас пароли хранятся в сыром виде в базе данных. Хранить их незашифрованными – дурной тон. В этой части мы добавим шифрование паролей.

Добавим в SecurityConfig шифрование:



Мы определили, что в качестве шифровальщика будет выступать BCryptPasswordEncoder, он является стандартным шифровальщиков приложений со Spring Security.

authenticationProvider.setPasswordEncoder(passwordEncoder()) «активирует» шифровальщик. Таким образом, при попытке авторизоваться, посланный в систему пароль будет сначала зашифрован с помощью этого шифровальщика, а потом сравнён с тем паролем, который лежит в базе.

Также добавим шифрование в UserServiceImpl при попытке регистрации.

Примем шифровальщик в конструкторе:



И добавим шифрование пароля перед попыткой регистрации:



Так как раньше у наших пользователей был пустой пароль, а выбранный шифровальщик пустой пароль шифрует непустым кодом, обновим пароли в data.sql:



Указанный здесь пароль – это именно то, что получается при попытке шифрования пустой строки.

Вот и всё, теперь все пароли зашифрованы.

Для того, чтобы обновить пароли трёх уже существующих пользователей, нужно их удалить из базы, либо удалить всю таблицу user.

Добавление возможности просмотра кода

Ну и последнее, что мы добавим в наш проект – это просмотр своего кода.

В `templates/contests/submissions/` добавим страницу с информацией о посылке:





Сразу же добавим в submission-list ссылку на решение. Для этого добавим следующий столбец сразу после Memory:



А также в блок, который строит саму таблицу, допишем такую строку:



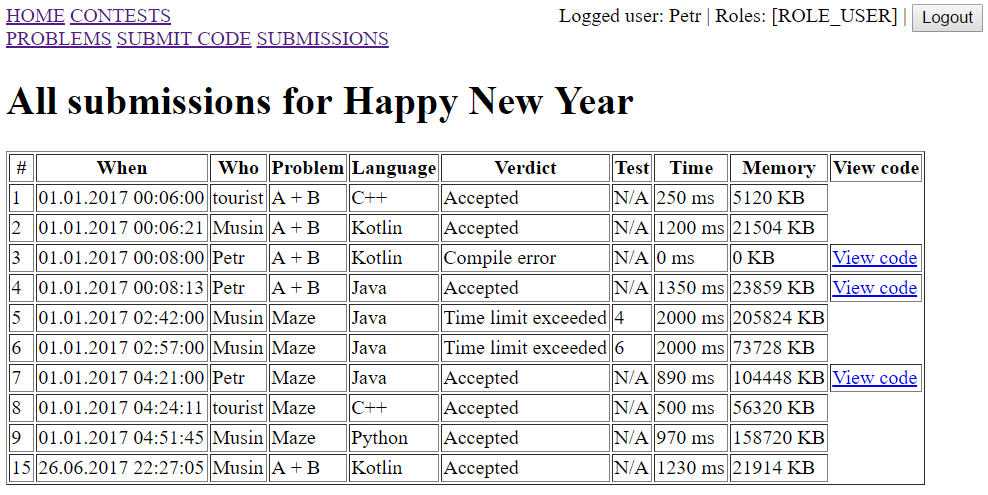
Таким образом, рядом со всеми посылками, сделанными текущим авторизованным пользователем, будет ссылка на просмотр решения.

Осталось лишь написать контроллер, который будет отвечать на эти запросы:

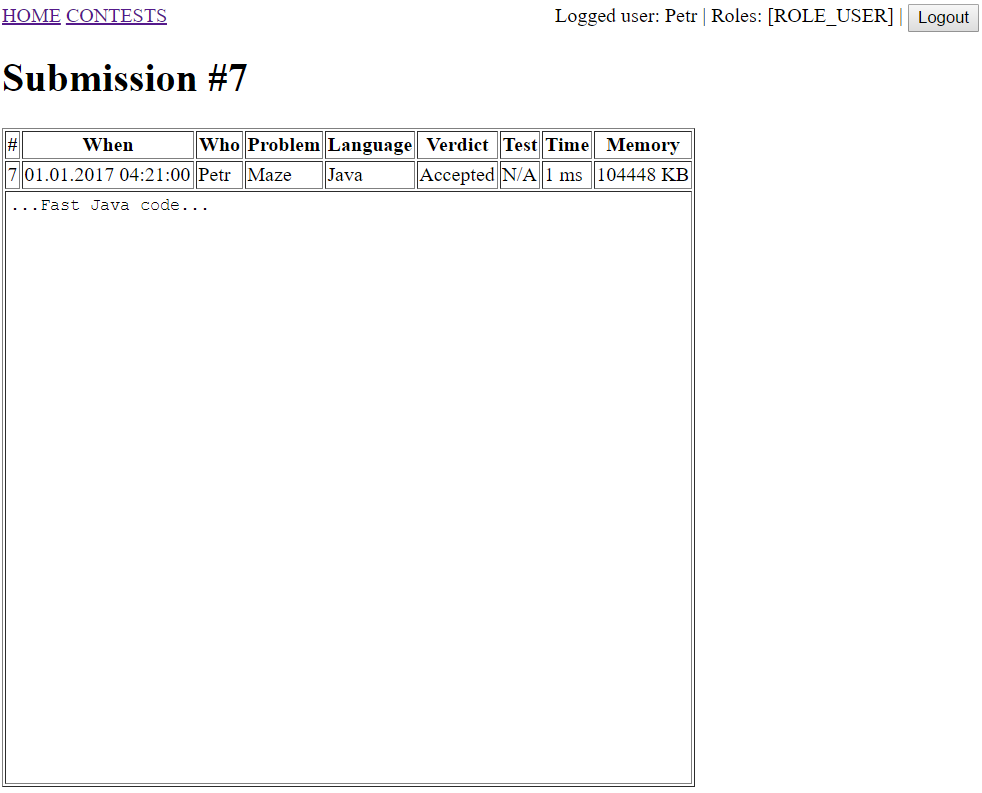


Проверяем.

Зайдём через пользователя Petr по адресу <http://localhost/contest/1/submissions>:



При нажатии, например, на последнюю ссылку «View code» получим следующее:



Итоги

Наше приложение научилось:

Работать с пользователями – авторизовывать и регистрировать их

Управлять контестами

Управлять задачами в контестах

Позволять отправлять решения задач и просматривать свой код

Полученное приложение можно развивать во множестве направлений.

К примеру, у него совсем не используются стили и весь UI построен на чистом html, так что на примере этого приложения можно оттачивать свои навыки front-end разработки.

Или, например, можно добавить пользователям личную информацию и позволить им общаться как на форуме.

Также можно разбить данное приложение на несколько частей и сделать микросервисную архитектуру, которая здорово поможет, если придётся масштабировать этот сервис.

Если у Вас появятся пожелания, предложения или Вы захотите сообщить об ошибке в руководстве или на GitHub – Вы можете сообщить мне об этом через GitHub.