Red Hat Decision Manager 7.8

Using Red Hat Business Optimizer with Spring Boot

Last Updated: 2020-09-07

Legal Notice

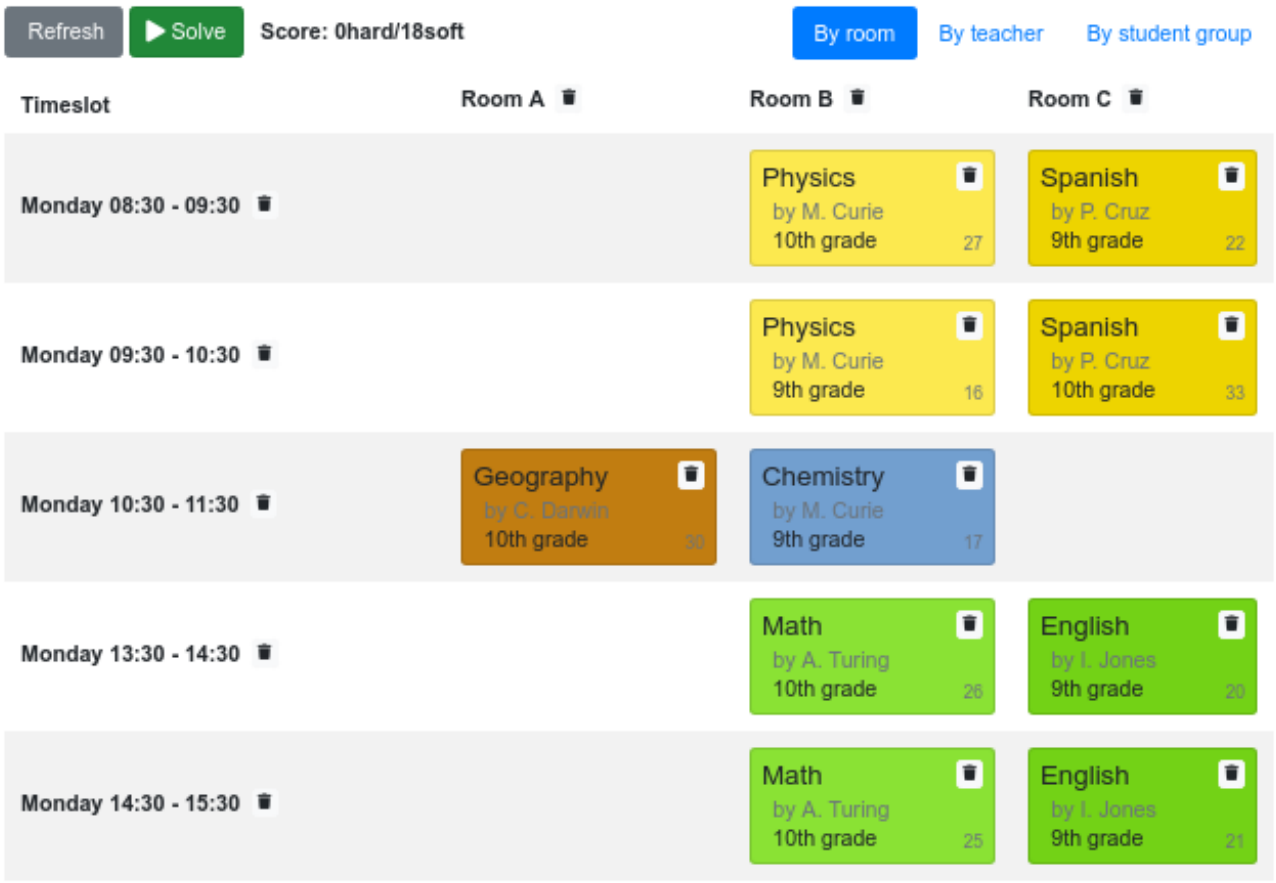
Abstract

This document describes how to design a Spring Boot application using Red Hat Business Optimizer.

Table of Contents

PREFACE

This guide walks you through the process of creating a Spring Boot application with Red Hat Business Optimizer's constraint solving artificial intelligence (AI). You will build a REST application that optimizes a school timetable for students and teachers.



Your service will assign **Lesson** instances to **Timeslot** and **Room** instances automatically by using AI to adhere to the following hard and soft *scheduling constraints*:

- A room can have at most one lesson at the same time.
- A teacher can teach at most one lesson at the same time.
- A student can attend at most one lesson at the same time.
- A teacher prefers to teach in a single room.
- A teacher prefers to teach sequential lessons and dislikes gaps between lessons.

Mathematically speaking, school timetabling is an *NP-hard* problem. That means it is difficult to scale. Simply iterating through all possible combinations with brute force would take millions of years for a non-trivial dataset, even on a supercomputer. Fortunately, AI constraint solvers such as Red Hat Business Optimizer have advanced algorithms that deliver a near-optimal solution in a reasonable amount of time. What is considered to be a reasonable amount of time is subjective and depends on the goals of your problem.

Prerequisites

- OpenJDK 8 or later is installed. Red Hat build of Open JDK is available from the Software Downloads page in the Red Hat Customer Portal (login required).

- Apache Maven 3.2 or higher or Gradle 4 or higher is installed. Maven is available from the Apache Maven Project website. Gradle is avaialbe from the Gradle Build Tool website.
- An IDE, such as IntelliJ IDEA, VSCode, Eclipse, or NetBeans is available.

CHAPTER 1. GENERATE THE RED HAT BUSINESS OPTIMIZER SPRING BOOT PROJECT

Spring Initialzr is a web-based application that creates a Spring Boot structure based on a few user inputs. You can use Spring Initializr to easily generate a Maven or Gradle Spring Boot project that you can customize for Red Hat Business Optimizer.

Procedure

1. Open Spring Initializr in a web browser:

   ```
   https://start.spring.io/
   ```

2. Select a project, language, Spring Boot version, and enter project metadata.

3. Click **Add Dependencies** and select **Spring Web** to add the **spring-boot-starter-web** dependency.

4. Click **GENERATE**. Your project ZIP file downloads.

5. Extract the ZIP file and change directory to your Spring Boot project directory.

6. Add the Red Hat Business Optimizer dependency (**optaplanner-spring-boot-starter**). Currently **optaplanner-spring-boot-starter** is not included in Spring Initializr so you must manually add it to your build file.

   - If you generated a Maven project, add the **optaplanner-spring-boot-starter** dependency to your project's **pom.xml**:

     ```
     <dependencyManagement>
       <dependencies>
         <dependency>
           <groupId>org.optaplanner</groupId>
           <artifactId>optaplanner-spring-boot-starter</artifactId>
           <version>{project-version}</version>
         </dependency>
       </dependencies>
     </dependencyManagement>
     ```

     The following example shows the **optaplanner-spring-boot-starter** dependency added to your project's **pom.xml** file:

     ```
     <?xml version="1.0" encoding="UTF-8"?>
     <project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
     http://maven.apache.org/xsd/maven-4.0.0.xsd">
       <modelVersion>4.0.0</modelVersion>
       <parent>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-parent</artifactId>
         <version>{version-org-spring-framework-boot}</version>
       </parent>
     ```

```
    <groupId>com.example</groupId>
    <artifactId>constraint-solving-ai-optaplanner</artifactId>
    <version>0.1.0-SNAPSHOT</version>
    <name>Constraint Solving AI with Red Hat Business Optimizer</name>
    <description>A Spring Boot Red Hat Business Optimizer example to generate a school
timetable.</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.optaplanner</groupId>
            <artifactId>optaplanner-spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.optaplanner</groupId>
                <artifactId>optaplanner-spring-boot-starter</artifactId>
                <version>{project-version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

- If you generated a Gradle project, add the **optaplanner-spring-boot-starter** to your **build.gradle** file:

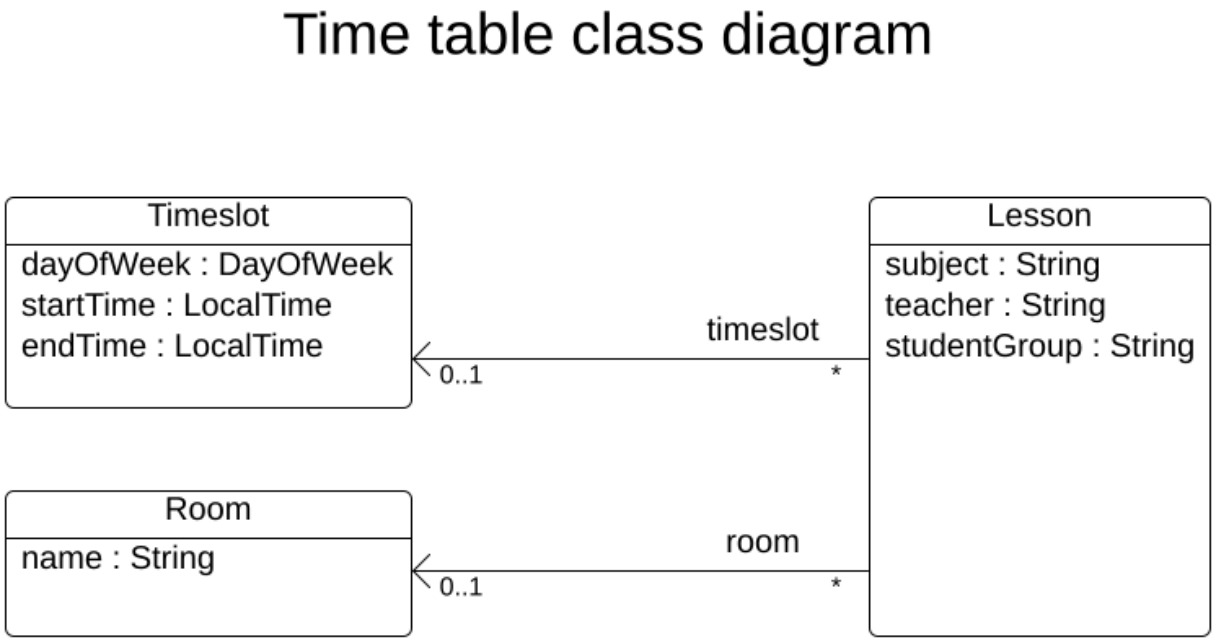
```
implementation "org.optaplanner:optaplanner-spring-boot-starter:{project-version}"
```

The following example shows the **optaplanner-spring-boot-starter** dependency added to your project's **build.gradle** file in a Gradle Java project:

```
plugins {
    id "org.springframework.boot" version "{version-org-spring-framework-boot}"
    id "io.spring.dependency-management" version "1.0.9.RELEASE"
    id "java"
}

group = "com.example"
version = "0.1.0-SNAPSHOT"
sourceCompatibility = "1.8"

repositories {
    mavenCentral()
}

dependencies {
    implementation "org.springframework.boot:spring-boot-starter-web"
    implementation "org.optaplanner:optaplanner-spring-boot-starter:{project-version}"
    testImplementation("org.springframework.boot:spring-boot-starter-test") {
        exclude group: "org.junit.vintage", module: "junit-vintage-engine"
    }
}

test {
    useJUnitPlatform()
}
```

CHAPTER 2. MODEL THE DOMAIN OBJECTS

The goal of the Red Hat Business Optimizer timetable project is to assign each lesson to a time slot and a room. To do this, add three classes, **Timeslot**, **Lesson**, and **Room**, as shown in the following diagram:



Timeslot

The **Timeslot** class represents a time interval when lessons are taught, for example, **Monday 10:30 - 11:30** or **Tuesday 13:30 - 14:30**. In this example, all time slots have the same duration and there are no time slots during lunch or other breaks.

A time slot has no date because a high school schedule just repeats every week. There is no need for continuous planning. A **Timeslot** is called a *problem fact* because no **Timeslot** instances change during solving. Such classes do not require any Red Hat Business Optimizer specific annotations.

Room

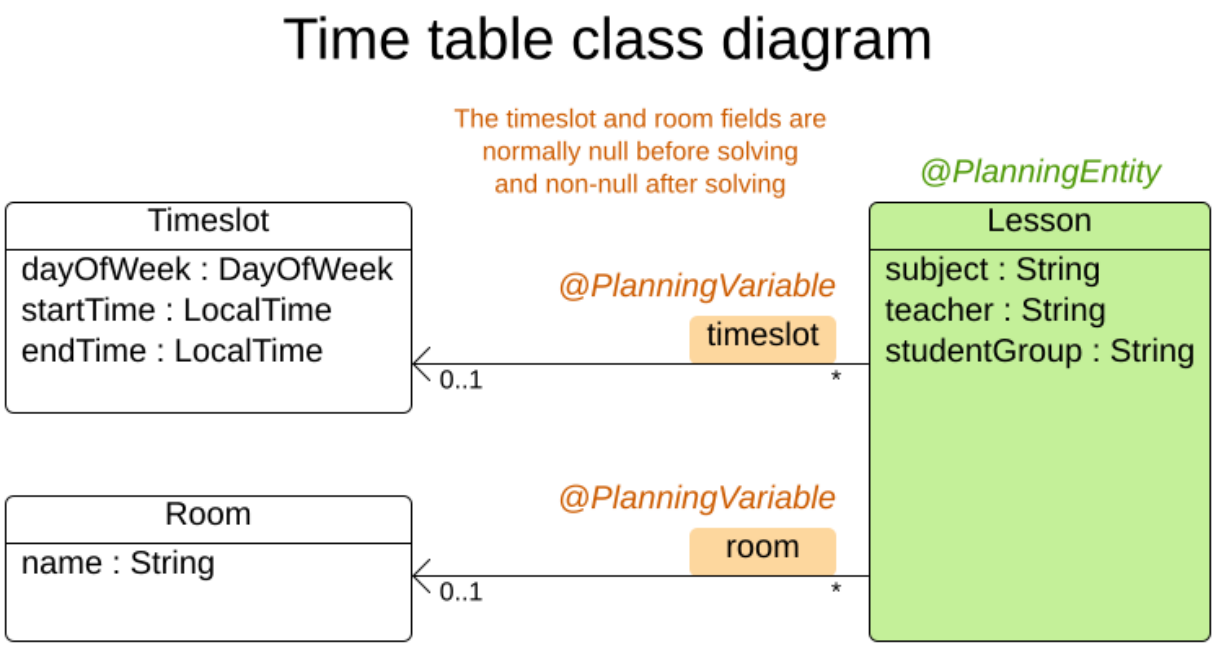
The **Room** class represents a location where lessons are taught, for example, **Room A** or **Room B**. In this example, all rooms are without capacity limits and they can accommodate all lessons.

Room instances do not change during solving so **Room** is also a *problem fact*.

Lesson

During a lesson, represented by the **Lesson** class, a teacher teaches a subject to a group of students, for example, **Math by A.Turing for 9th grade** or **Chemistry by M.Curie for 10th grade**. If a subject is taught multiple times each week by the same teacher to the same student group, there are multiple **Lesson** instances that are only distinguishable by **id**. For example, the 9th grade has six math lessons a week.

During solving, Red Hat Business Optimizer changes the **timeslot** and **room** fields of the **Lesson** class to assign each lesson to a time slot and a room. Because Red Hat Business Optimizer changes these fields, **Lesson** is a *planning entity*:



Most of the fields in the previous diagram contain input data, except for the orange fields. A lesson's **timeslot** and **room** fields are unassigned (**null**) in the input data and assigned (not **null**) in the output data. Red Hat Business Optimizer changes these fields during solving. Such fields are called planning variables. In order for Red Hat Business Optimizer to recognize them, both the **timeslot** and **room** fields require an **@PlanningVariable** annotation. Their containing class, **Lesson**, requires an **@PlanningEntity** annotation.

Procedure

1. Create the **src/main/java/com/example/domain/Timeslot.java** class:

```
package com.example.domain;

import java.time.DayOfWeek;
import java.time.LocalTime;

public class Timeslot {

    private DayOfWeek dayOfWeek;
    private LocalTime startTime;
    private LocalTime endTime;

    private Timeslot() {
    }

    public Timeslot(DayOfWeek dayOfWeek, LocalTime startTime, LocalTime endTime) {
        this.dayOfWeek = dayOfWeek;
        this.startTime = startTime;
        this.endTime = endTime;
    }

    @Override
    public String toString() {
        return dayOfWeek + " " + startTime.toString();
```

```
    }

    // ********************************
    // Getters and setters
    // ********************************

    public DayOfWeek getDayOfWeek() {
        return dayOfWeek;
    }

    public LocalTime getStartTime() {
        return startTime;
    }

    public LocalTime getEndTime() {
        return endTime;
    }

}
```

Notice the **toString()** method keeps the output short so it is easier to read Red Hat Business Optimizer's **DEBUG** or **TRACE** log, as shown later.

2. Create the **src/main/java/com/example/domain/Room.java** class:

```
package com.example.domain;

public class Room {

    private String name;

    private Room() {
    }

    public Room(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }

    // ********************************
    // Getters and setters
    // ********************************

    public String getName() {
        return name;
    }

}
```

3. Create the **src/main/java/com/example/domain/Lesson.java** class:

```
package com.example.domain;
```

```
import org.optaplanner.core.api.domain.entity.PlanningEntity;
import org.optaplanner.core.api.domain.variable.PlanningVariable;

@PlanningEntity
public class Lesson {

    private Long id;

    private String subject;
    private String teacher;
    private String studentGroup;

    @PlanningVariable(valueRangeProviderRefs = "timeslotRange")
    private Timeslot timeslot;

    @PlanningVariable(valueRangeProviderRefs = "roomRange")
    private Room room;

    private Lesson() {
    }

    public Lesson(Long id, String subject, String teacher, String studentGroup) {
        this.id = id;
        this.subject = subject;
        this.teacher = teacher;
        this.studentGroup = studentGroup;
    }

    @Override
    public String toString() {
        return subject + "(" + id + ")";
    }

    // ********************************
    // Getters and setters
    // ********************************

    public Long getId() {
        return id;
    }

    public String getSubject() {
        return subject;
    }

    public String getTeacher() {
        return teacher;
    }

    public String getStudentGroup() {
        return studentGroup;
    }

    public Timeslot getTimeslot() {
        return timeslot;
```

```
    }

    public void setTimeslot(Timeslot timeslot) {
        this.timeslot = timeslot;
    }

    public Room getRoom() {
        return room;
    }

    public void setRoom(Room room) {
        this.room = room;
    }

}
```

The **Lesson** class has an **@PlanningEntity** annotation, so Red Hat Business Optimizer knows that this class changes during solving because it contains one or more planning variables.

The **timeslot** field has an **@PlanningVariable** annotation, so Red Hat Business Optimizer knows that it can change its value. In order to find potential **Timeslot** instances to assign to this field, Red Hat Business Optimizer uses the **valueRangeProviderRefs** property to connect to a value range provider that provides a **List<Timeslot>** to pick from. See Chapter 4, *Gather the domain objects in a planning solution* for information about value range providers.

The **room** field also has an **@PlanningVariable** annotation for the same reasons.

CHAPTER 3. DEFINE THE CONSTRAINTS AND CALCULATE THE SCORE

When solving a problem, a *score* represents the quality of a given solution. The higher the score the better. Red Hat Business Optimizer looks for the best solution, which is the solution with the highest score found in the available time. It might be the *optimal* solution.

Because the timetable example use case has hard and soft constraints, use the **HardSoftScore** class to represent the score:

- Hard constraints must not be broken. For example: *A room can have at most one lesson at the same time.*
- Soft constraints should not be broken. For example: *A teacher prefers to teach in a single room.*

Hard constraints are weighted against other hard constraints. Soft constraints are weighted against other soft constraints. Hard constraints always outweigh soft constraints, regardless of their respective weights.

To calculate the score, you could implement an **EasyScoreCalculator** class:

```
public class TimeTableEasyScoreCalculator implements EasyScoreCalculator<TimeTable> {

    @Override
    public HardSoftScore calculateScore(TimeTable timeTable) {
        List<Lesson> lessonList = timeTable.getLessonList();
        int hardScore = 0;
        for (Lesson a : lessonList) {
            for (Lesson b : lessonList) {
                if (a.getTimeslot() != null && a.getTimeslot().equals(b.getTimeslot())
                        && a.getId() < b.getId()) {
                    // A room can accommodate at most one lesson at the same time.
                    if (a.getRoom() != null && a.getRoom().equals(b.getRoom())) {
                        hardScore--;
                    }
                    // A teacher can teach at most one lesson at the same time.
                    if (a.getTeacher().equals(b.getTeacher())) {
                        hardScore--;
                    }
                    // A student can attend at most one lesson at the same time.
                    if (a.getStudentGroup().equals(b.getStudentGroup())) {
                        hardScore--;
                    }
                }
            }
        }
        int softScore = 0;
        // Soft constraints are only implemented in the "complete" implementation
        return HardSoftScore.of(hardScore, softScore);
    }

}
```

Unfortunately, this solution does not scale well because it is non-incremental: every time a lesson is assigned to a different time slot or room, all lessons are re-evaluated to calculate the new score.

A better solution is to create a **src/main/java/com/example/solver/TimeTableConstraintProvider.java** class to perform incremental score calculation. This class uses Red Hat Business Optimizer's ConstraintStream API which is inspired by Java 8 Streams and SQL. The **ConstraintProvider** scales an order of magnitude better than the **EasyScoreCalculator**: $O(n)$ instead of $O(n^2)$.

Procedure

Create the following **src/main/java/com/example/solver/TimeTableConstraintProvider.java** class:

```
package com.example.solver;

import com.example.domain.Lesson;
import org.optaplanner.core.api.score.buildin.hardsoft.HardSoftScore;
import org.optaplanner.core.api.score.stream.Constraint;
import org.optaplanner.core.api.score.stream.ConstraintFactory;
import org.optaplanner.core.api.score.stream.ConstraintProvider;
import org.optaplanner.core.api.score.stream.Joiners;

public class TimeTableConstraintProvider implements ConstraintProvider {

    @Override
    public Constraint[] defineConstraints(ConstraintFactory constraintFactory) {
        return new Constraint[] {
                // Hard constraints
                roomConflict(constraintFactory),
                teacherConflict(constraintFactory),
                studentGroupConflict(constraintFactory),
                // Soft constraints are only implemented in the "complete" implementation
        };
    }

    private Constraint roomConflict(ConstraintFactory constraintFactory) {
        // A room can accommodate at most one lesson at the same time.

        // Select a lesson ...
        return constraintFactory.from(Lesson.class)
                // ... and pair it with another lesson ...
                .join(Lesson.class,
                        // ... in the same timeslot ...
                        Joiners.equal(Lesson::getTimeslot),
                        // ... in the same room ...
                        Joiners.equal(Lesson::getRoom),
                        // ... and the pair is unique (different id, no reverse pairs)
                        Joiners.lessThan(Lesson::getId))
                // then penalize each pair with a hard weight.
                .penalize("Room conflict", HardSoftScore.ONE_HARD);
    }

    private Constraint teacherConflict(ConstraintFactory constraintFactory) {
        // A teacher can teach at most one lesson at the same time.
        return constraintFactory.from(Lesson.class)
                .join(Lesson.class,
                        Joiners.equal(Lesson::getTimeslot),
                        Joiners.equal(Lesson::getTeacher),
                        Joiners.lessThan(Lesson::getId))
                .penalize("Teacher conflict", HardSoftScore.ONE_HARD);
```

```
    }

    private Constraint studentGroupConflict(ConstraintFactory constraintFactory) {
        // A student can attend at most one lesson at the same time.
        return constraintFactory.from(Lesson.class)
                .join(Lesson.class,
                        Joiners.equal(Lesson::getTimeslot),
                        Joiners.equal(Lesson::getStudentGroup),
                        Joiners.lessThan(Lesson::getId))
                .penalize("Student group conflict", HardSoftScore.ONE_HARD);
    }

}
```

CHAPTER 4. GATHER THE DOMAIN OBJECTS IN A PLANNING SOLUTION

A **TimeTable** instance wraps all **Timeslot**, **Room**, and **Lesson** instances of a single dataset. Furthermore, because it contains all lessons, each with a specific planning variable state, it is a *planning solution* and it has a score:

- If lessons are still unassigned, then it is an *uninitialized* solution, for example, a solution with the score **-4init/0hard/0soft**.
- If it breaks hard constraints, then it is an *infeasible* solution, for example, a solution with the score **-2hard/-3soft**.
- If it adheres to all hard constraints, then it is a *feasible* solution, for example, a solution with the score **0hard/-7soft**.

The **TimeTable** class has an **@PlanningSolution** annotation, so Red Hat Business Optimizer knows that this class contains all of the input and output data.

Specifically, this class is the input of the problem:

- A **timeslotList** field with all time slots
  - This is a list of problem facts, because they do not change during solving.
- A **roomList** field with all rooms
  - This is a list of problem facts, because they do not change during solving.
- A **lessonList** field with all lessons
  - This is a list of planning entities because they change during solving.
  - Of each **Lesson**:
    - The values of the **timeslot** and **room** fields are typically still **null**, so unassigned. They are planning variables.
    - The other fields, such as **subject**, **teacher** and **studentGroup**, are filled in. These fields are problem properties.

However, this class is also the output of the solution:

- A **lessonList** field for which each **Lesson** instance has non-null **timeslot** and **room** fields after solving
- A **score** field that represents the quality of the output solution, for example, **0hard/-5soft**

Procedure

Create the **src/main/java/com/example/domain/TimeTable.java** class:

```
package com.example.domain;

import java.util.List;

import org.optaplanner.core.api.domain.solution.PlanningEntityCollectionProperty;
```

```
import org.optaplanner.core.api.domain.solution.PlanningScore;
import org.optaplanner.core.api.domain.solution.PlanningSolution;
import org.optaplanner.core.api.domain.solution.drools.ProblemFactCollectionProperty;
import org.optaplanner.core.api.domain.valuerange.ValueRangeProvider;
import org.optaplanner.core.api.score.buildin.hardsoft.HardSoftScore;

@PlanningSolution
public class TimeTable {

    @ValueRangeProvider(id = "timeslotRange")
    @ProblemFactCollectionProperty
    private List<Timeslot> timeslotList;

    @ValueRangeProvider(id = "roomRange")
    @ProblemFactCollectionProperty
    private List<Room> roomList;

    @PlanningEntityCollectionProperty
    private List<Lesson> lessonList;

    @PlanningScore
    private HardSoftScore score;

    private TimeTable() {
    }

    public TimeTable(List<Timeslot> timeslotList, List<Room> roomList,
            List<Lesson> lessonList) {
        this.timeslotList = timeslotList;
        this.roomList = roomList;
        this.lessonList = lessonList;
    }

    // ********************************
    // Getters and setters
    // ********************************

    public List<Timeslot> getTimeslotList() {
        return timeslotList;
    }

    public List<Room> getRoomList() {
        return roomList;
    }

    public List<Lesson> getLessonList() {
        return lessonList;
    }

    public HardSoftScore getScore() {
        return score;
    }

}
```

The value range providers

The **timeslotList** field is a value range provider. It holds the **Timeslot** instances which Red Hat Business Optimizer can pick from to assign to the **timeslot** field of **Lesson** instances. The **timeslotList** field has an **@ValueRangeProvider** annotation to connect those two, by matching the **id** with the **valueRangeProviderRefs** of the **@PlanningVariable** in the **Lesson**.

Following the same logic, the **roomList** field also has an **@ValueRangeProvider** annotation.

The problem fact and planning entity properties

Furthermore, Red Hat Business Optimizer needs to know which **Lesson** instances it can change as well as how to retrieve the **Timeslot** and **Room** instances used for score calculation by your **TimeTableConstraintProvider**.

The **timeslotList** and **roomList** fields have an **@ProblemFactCollectionProperty** annotation, so your **TimeTableConstraintProvider** can select from those instances.

The **lessonList** has an **@PlanningEntityCollectionProperty** annotation, so Red Hat Business Optimizer can change them during solving and your **TimeTableConstraintProvider** can select from those too.

CHAPTER 5. CREATE THE TIMETABLE SERVICE

Now you are ready to put everything together and create a REST service. But solving planning problems on REST threads causes HTTP timeout issues. Therefore, the Spring Boot starter injects a **SolverManager**, which runs solvers in a separate thread pool and can solve multiple datasets in parallel.

Procedure

Create the **src/main/java/com/example/solver/TimeTableController.java** class:

```
package com.example.solver;

import java.util.UUID;
import java.util.concurrent.ExecutionException;

import com.example.domain.TimeTable;
import org.optaplanner.core.api.solver.SolverJob;
import org.optaplanner.core.api.solver.SolverManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/timeTable")
public class TimeTableController {

    @Autowired
    private SolverManager<TimeTable, UUID> solverManager;

    @PostMapping("/solve")
    public TimeTable solve(@RequestBody TimeTable problem) {
        UUID problemId = UUID.randomUUID();
        // Submit the problem to start solving
        SolverJob<TimeTable, UUID> solverJob = solverManager.solve(problemId, problem);
        TimeTable solution;
        try {
            // Wait until the solving ends
            solution = solverJob.getFinalBestSolution();
        } catch (InterruptedException | ExecutionException e) {
            throw new IllegalStateException("Solving failed.", e);
        }
        return solution;
    }

}
```

In this example, the initial implementation waits for the solver to finish, which can still cause an HTTP timeout. The *complete* implementation avoids HTTP timeouts much more elegantly.

CHAPTER 6. SET THE SOLVER TERMINATION TIME

If your planning application does not have a termination setting or a termination event, it theoretically runs forever and in reality eventually causes an HTTP timeout error. To prevent this from occurring, use the **optaplanner.solver.termination.spent-limit** parameter to specify the length of time after which the application terminates. In most applications, set the time to at least five minutes (**5m**). However, in the Timetable example, limit the solving time to five second, which is short enough to avoid the HTTP timeout.

Procedure

Create the **src/main/resources/application.properties** file with the following content:

```
optaplanner.solver.termination.spent-limit=5s
```

CHAPTER 7. MAKE THE APPLICATION EXECUTABLE

After you complete the Red Hat Business Optimizer Spring Boot timetable project, package everything into a single executable JAR file driven by a standard Java **main()** method.

Prerequisites

- You have a completed Red Hat Business Optimizer Spring Boot timetable project.

Procedure

1. Create the **TimeTableSpringBootApp.java** class with the following content:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TimeTableSpringBootApp {

    public static void main(String[] args) {
        SpringApplication.run(TimeTableSpringBootApp.class, args);
    }

}
```

2. Replace the **src/main/java/com/example/DemoApplication.java** class created by Spring Initializr with the **TimeTableSpringBootApp.java** class.

3. Run the **TimeTableSpringBootApp.java** class as the main class of a regular Java application.

7.1. TRY THE TIMETABLE APPLICATION

After you start the Red Hat Business Optimizer Spring Boot timetable application, you can test the REST service with any REST client that you want. This example uses the Linux **curl** command to send a POST request.

Prerequisites

- The Red Hat Business Optimizer Spring Boot timetable application is running.

Procedure

Enter the following command:

```
$ curl -i -X POST http://localhost:8080/timeTable/solve -H "Content-Type:application/json" -d
'{"timeslotList":[{"dayOfWeek":"MONDAY","startTime":"08:30:00","endTime":"09:30:00"},
{"dayOfWeek":"MONDAY","startTime":"09:30:00","endTime":"10:30:00"}],"roomList":[{"name":"Room
A"},{"name":"Room B"}],"lessonList":[{"id":1,"subject":"Math","teacher":"A. Turing","studentGroup":"9th
grade"},{"id":2,"subject":"Chemistry","teacher":"M. Curie","studentGroup":"9th grade"},
{"id":3,"subject":"French","teacher":"M. Curie","studentGroup":"10th grade"},
{"id":4,"subject":"History","teacher":"I. Jones","studentGroup":"10th grade"}]}'
```

After about five seconds, the termination spent time defined in **application.properties**, the service returns an output similar to the following example:

```
HTTP/1.1 200
Content-Type: application/json
...

{"timeslotList":...,"roomList":...,"lessonList":[{"id":1,"subject":"Math","teacher":"A.
Turing","studentGroup":"9th grade","timeslot":
{"dayOfWeek":"MONDAY","startTime":"08:30:00","endTime":"09:30:00"},"room":{"name":"Room A"}},
{"id":2,"subject":"Chemistry","teacher":"M. Curie","studentGroup":"9th grade","timeslot":
{"dayOfWeek":"MONDAY","startTime":"09:30:00","endTime":"10:30:00"},"room":{"name":"Room A"}},
{"id":3,"subject":"French","teacher":"M. Curie","studentGroup":"10th grade","timeslot":
{"dayOfWeek":"MONDAY","startTime":"08:30:00","endTime":"09:30:00"},"room":{"name":"Room B"}},
{"id":4,"subject":"History","teacher":"I. Jones","studentGroup":"10th grade","timeslot":
{"dayOfWeek":"MONDAY","startTime":"09:30:00","endTime":"10:30:00"},"room":{"name":"Room
B"}}],"score":"0hard/0soft"}
```

Notice that the application assigned all four lessons to one of the two time slots and one of the two rooms. Also notice that it conforms to all hard constraints. For example, M. Curie's two lessons are in different time slots.

On the server side, the **info** log shows what Red Hat Business Optimizer did in those five seconds:

```
... Solving started: time spent (33), best score (-8init/0hard/0soft), environment mode
(REPRODUCIBLE), random (JDK with seed 0).
... Construction Heuristic phase (0) ended: time spent (73), best score (0hard/0soft), score calculation
speed (459/sec), step total (4).
... Local Search phase (1) ended: time spent (5000), best score (0hard/0soft), score calculation
speed (28949/sec), step total (28398).
... Solving ended: time spent (5000), best score (0hard/0soft), score calculation speed (28524/sec),
phase total (2), environment mode (REPRODUCIBLE).
```

7.2. TEST THE APPLICATION

A good application includes test coverage. This example tests the Timetable Red Hat Business Optimizer Spring Boot application. It uses a JUnit test to generate a test dataset and send it to the **TimeTableController** to solve.

Procedure

Create the **src/test/java/com/example/solver/TimeTableControllerTest.java** class with the following content:

```
package com.example.solver;

import java.time.DayOfWeek;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.List;

import com.example.domain.Lesson;
import com.example.domain.Room;
import com.example.domain.TimeTable;
import com.example.domain.Timeslot;
```

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;

@SpringBootTest(properties = {
        "optaplanner.solver.termination.spent-limit=1h", // Effectively disable this termination in favor of
the best-score-limit
        "optaplanner.solver.termination.best-score-limit=0hard/*soft"})
public class TimeTableControllerTest {

    @Autowired
    private TimeTableController timeTableController;

    @Test
    @Timeout(600_000)
    public void solve() {
        TimeTable problem = generateProblem();
        TimeTable solution = timeTableController.solve(problem);
        assertFalse(solution.getLessonList().isEmpty());
        for (Lesson lesson : solution.getLessonList()) {
            assertNotNull(lesson.getTimeslot());
            assertNotNull(lesson.getRoom());
        }
        assertTrue(solution.getScore().isFeasible());
    }

    private TimeTable generateProblem() {
        List<Timeslot> timeslotList = new ArrayList<>();
        timeslotList.add(new Timeslot(DayOfWeek.MONDAY, LocalTime.of(8, 30), LocalTime.of(9,
30)));
        timeslotList.add(new Timeslot(DayOfWeek.MONDAY, LocalTime.of(9, 30), LocalTime.of(10,
30)));
        timeslotList.add(new Timeslot(DayOfWeek.MONDAY, LocalTime.of(10, 30), LocalTime.of(11,
30)));
        timeslotList.add(new Timeslot(DayOfWeek.MONDAY, LocalTime.of(13, 30), LocalTime.of(14,
30)));
        timeslotList.add(new Timeslot(DayOfWeek.MONDAY, LocalTime.of(14, 30), LocalTime.of(15,
30)));

        List<Room> roomList = new ArrayList<>();
        roomList.add(new Room("Room A"));
        roomList.add(new Room("Room B"));
        roomList.add(new Room("Room C"));

        List<Lesson> lessonList = new ArrayList<>();
        lessonList.add(new Lesson(101L, "Math", "B. May", "9th grade"));
        lessonList.add(new Lesson(102L, "Physics", "M. Curie", "9th grade"));
        lessonList.add(new Lesson(103L, "Geography", "M. Polo", "9th grade"));
        lessonList.add(new Lesson(104L, "English", "I. Jones", "9th grade"));
        lessonList.add(new Lesson(105L, "Spanish", "P. Cruz", "9th grade"));
```

```
        lessonList.add(new Lesson(201L, "Math", "B. May", "10th grade"));
        lessonList.add(new Lesson(202L, "Chemistry", "M. Curie", "10th grade"));
        lessonList.add(new Lesson(203L, "History", "I. Jones", "10th grade"));
        lessonList.add(new Lesson(204L, "English", "P. Cruz", "10th grade"));
        lessonList.add(new Lesson(205L, "French", "M. Curie", "10th grade"));
        return new TimeTable(timeslotList, roomList, lessonList);
    }

}
```

This test verifies that after solving, all lessons are assigned to a time slot and a room. It also verifies that it found a feasible solution (no hard constraints broken).

Normally, the solver finds a feasible solution in less than 200 milliseconds. Notice how the **@SpringBootTest** annotation's **properties** overwrites the solver termination to terminate as soon as a feasible solution (**0hard/*soft**) is found. This avoids hard coding a solver time, because the unit test might run on arbitrary hardware. This approach ensures that the test runs long enough to find a feasible solution, even on slow machines. However, it does not run a millisecond longer than it strictly must, even on fast machines.

7.3. LOGGING

After you complete the Red Hat Business Optimizer Spring Boot timetable application, you can use logging information to help you fine-tune the constraints in the **ConstraintProvider**. Review the score calculation speed in the **info** log file to assess the impact of changes to your constraints. Run the application in debug mode to show every step that your application takes or use trace logging to log every step and every move.

Procedure

1. Run the timetable application for a fixed amount of time, for example, five minutes.
2. Review the score calculation speed in the **log** file as shown in the following example:

```
... Solving ended: ..., score calculation speed (29455/sec), ...
```

3. Change a constraint, run the planning application again for the same amount of time, and review the score calculation speed recorded in the **log** file.
4. Run the application in debug mode to log every step:
   - To run debug mode from the command line, use the **-D** system property.
   - To change logging in the **application.properties** file, add the following line to that file:

```
logging.level.org.optaplanner=debug
```

     The following example shows output in the **log** file in debug mode:

```
... Solving started: time spent (67), best score (-20init/0hard/0soft), environment mode
(REPRODUCIBLE), random (JDK with seed 0).
...      CH step (0), time spent (128), score (-18init/0hard/0soft), selected move count (15),
picked move ([Math(101) {null -> Room A}, Math(101) {null -> MONDAY 08:30}]).
```

```
...     CH step (1), time spent (145), score (-16init/0hard/0soft), selected move count (15),
picked move ([Physics(102) {null -> Room A}, Physics(102) {null -> MONDAY 09:30}]).
...
```

5. Use **trace** logging to show every step and every move per step.

CHAPTER 8. ADD DATABASE AND UI INTEGRATION

After you create the Red Hat Business Optimizer application example with Spring Boot, add database and UI integration.

Prerequisite

- You have created the Red Hat Business Optimizer Spring Boot timetable example.

Procedure

1. Create Java Persistence API (JPA) repositories for **Timeslot**, **Room**, and **Lesson**. For information about creating JPA repositories, see *Accessing Data with JPA* on the Spring website.
2. Expose the JPA repositories through REST. For information about exposing the repositories, see *Accessing JPA Data with REST* on the Spring website.
3. Build a **TimeTableRepository** facade to read and write a **TimeTable** in a single transaction.
4. Adjust the **TimeTableController** as shown in the following example:

```
package com.example.solver;

import com.example.domain.TimeTable;
import com.example.persistence.TimeTableRepository;
import org.optaplanner.core.api.score.ScoreManager;
import org.optaplanner.core.api.solver.SolverManager;
import org.optaplanner.core.api.solver.SolverStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/timeTable")
public class TimeTableController {

    @Autowired
    private TimeTableRepository timeTableRepository;
    @Autowired
    private SolverManager<TimeTable, Long> solverManager;
    @Autowired
    private ScoreManager<TimeTable> scoreManager;

    // To try, GET http://localhost:8080/timeTable
    @GetMapping()
    public TimeTable getTimeTable() {
        // Get the solver status before loading the solution
        // to avoid the race condition that the solver terminates between them
        SolverStatus solverStatus = getSolverStatus();
        TimeTable solution =
timeTableRepository.findById(TimeTableRepository.SINGLETON_TIME_TABLE_ID);
        scoreManager.updateScore(solution); // Sets the score
```

```
        solution.setSolverStatus(solverStatus);
        return solution;
    }

    @PostMapping("/solve")
    public void solve() {
        solverManager.solveAndListen(TimeTableRepository.SINGLETON_TIME_TABLE_ID,
                timeTableRepository::findById,
                timeTableRepository::save);
    }

    public SolverStatus getSolverStatus() {
        return
solverManager.getSolverStatus(TimeTableRepository.SINGLETON_TIME_TABLE_ID);
    }

    @PostMapping("/stopSolving")
    public void stopSolving() {
        solverManager.terminateEarly(TimeTableRepository.SINGLETON_TIME_TABLE_ID);
    }

}
```

For simplicity, this code handles only one **TimeTable** instance, but it is straightforward to enable multi-tenancy and handle multiple **TimeTable** instances of different high schools in parallel.

The **getTimeTable()** method returns the latest timetable from the database. It uses the **ScoreManager** (which is automatically injected) to calculate the score of that timetable so the UI can show the score.

The **solve()** method starts a job to solve the current timetable and store the time slot and room assignments in the database. It uses the **SolverManager.solveAndListen()** method to listen to intermediate best solutions and update the database accordingly. This enables the UI to show progress while the backend is still solving.

5. Now that the **solve()** method returns immediately, adjust the **TimeTableControllerTest** as shown in the following example:

```
package com.example.solver;

import com.example.domain.Lesson;
import com.example.domain.TimeTable;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import org.optaplanner.core.api.solver.SolverStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;

@SpringBootTest(properties = {
        "optaplanner.solver.termination.spent-limit=1h", // Effectively disable this termination in
favor of the best-score-limit
        "optaplanner.solver.termination.best-score-limit=0hard/*soft"})
```

```
public class TimeTableControllerTest {

    @Autowired
    private TimeTableController timeTableController;

    @Test
    @Timeout(600_000)
    public void solveDemoDataUntilFeasible() throws InterruptedException {
        timeTableController.solve();
        TimeTable timeTable = timeTableController.getTimeTable();
        while (timeTable.getSolverStatus() != SolverStatus.NOT_SOLVING) {
            // Quick polling (not a Test Thread Sleep anti-pattern)
            // Test is still fast on fast machines and doesn't randomly fail on slow machines.
            Thread.sleep(20L);
            timeTable = timeTableController.getTimeTable();
        }
        assertFalse(timeTable.getLessonList().isEmpty());
        for (Lesson lesson : timeTable.getLessonList()) {
            assertNotNull(lesson.getTimeslot());
            assertNotNull(lesson.getRoom());
        }
        assertTrue(timeTable.getScore().isFeasible());
    }

}
```

6. Poll for the latest solution until the solver finishes solving.

7. To visualize the timetable, build an attractive web UI on top of these REST methods.

APPENDIX A. VERSIONING INFORMATION

Documentation last updated on Thursday, September 08, 2020.