



Red Hat Decision Manager 7.8

Spring boot との Red Hat Business Optimizer の使用

ガイド

Red Hat Decision Manager 7.8 Spring boot との Red Hat Business Optimizer の使用

ガイド

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Using_Red_Hat_Business_Optimizer_with_Spring_Boot.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

このドキュメントでは、Red Hat Business Optimizer を使用して Spring Boot アプリケーションを設計する方法について説明します。

目次

はじめに	3
第1章 RED HAT BUSINESS OPTIMIZER SPRING BOOT プロジェクトの生成	5
第2章 ドメインオブジェクトのモデル化	8
第3章 制約の定義およびスコアの計算	13
第4章 プランニングソリューションでのドメインオブジェクトの収集	16
第5章 TIMETABLE サービスの作成	19
第6章 ソルバー終了時間の設定	20
第7章 アプリケーションを実行可能にする手順	21
7.1. 時間割アプリケーションの試行	21
7.2. アプリケーションのテスト	22
7.3. ログイン	24
第8章 データベースと UI 統合の追加	26
付録A バージョン情報	29

はじめに

本書では、Red Hat Business Optimizer の制約解決人工知能 (AI) を使用して Spring Boot アプリケーションもを作成するプロセスを説明します。学生および教師向けの時間割を最適化する REST アプリケーションを構築していきます。

Refresh	Solve	Score: 0hard/18soft	By room	By teacher	By student group
Timeslot	Room A	Room B	Room C		
Monday 08:30 - 09:30		Physics by M. Curie 10th grade 27	Spanish by P. Cruz 9th grade 22		
Monday 09:30 - 10:30		Physics by M. Curie 9th grade 16	Spanish by P. Cruz 10th grade 33		
Monday 10:30 - 11:30	Geography by C. Darwin 10th grade 30	Chemistry by M. Curie 9th grade 17			
Monday 13:30 - 14:30		Math by A. Turing 10th grade 26	English by I. Jones 9th grade 20		
Monday 14:30 - 15:30		Math by A. Turing 10th grade 25	English by I. Jones 9th grade 21		

サービスは、AI を使用して、以下のハードおよびソフトの **スケジュール制約** に準拠し、**Lesson** インスタンスを **Timeslot** インスタンスと **Room** インスタンスに自動的に割り当てます。

- 1部屋に同時に割り当てることができる授業は、最大1コマです。
- 教師が同時に一度に行うことができる授業は最大1回です。
- 生徒は同時に出席できる授業は最大1コマです。
- 教師は、1つの部屋での授業を希望します。
- 教師は、連続した授業を好み、授業間に時間が空くのを嫌います。

数学的に考えると、学校の時間割は **NP 困難** の問題であります。つまり、スケーリングが困難です。総当たり攻撃で考えられる組み合わせを単純にすべて反復すると、スーパーコンピュータを使用したとしても、非自明的なデータセットを取得するのに数百年かかります。幸い、Red Hat Business Optimizer などの AI 制約ソルバーには、妥当な時間内にほぼ最適なソリューションを提供する高度なアルゴリズムがあります。妥当な期間として考慮される内容は、問題の目的によって異なります。

前提条件

- OpenJDK 8 以降がインストールされている。Red Hat ビルドの Open JDK は Red Hat カスタマーポータル (ログインが必要) の [ソフトウェアダウンロード](#) ページから入手できます。

- Apache Maven 3.2 以降または Gradle 4 以降がインストールされている。Maven は [Apache Maven Project](#) の Web サイトから入手できます。Gradle は、[Gradle Build Tool](#) の Web サイトから利用できます。
- IntelliJ IDEA、VSCode、Eclipse、NetBeans などの IDE が利用できる。

第1章 RED HAT BUSINESS OPTIMIZER SPRING BOOT プロジェクトの生成

Spring Initializr は Web ベースのアプリケーションで、少数のユーザー入力をもとに、Spring Boot 構造を作成します。Spring Initializr を使用して簡単に Maven または Gradle Spring Boot プロジェクトを生成し、Red Hat Business Optimizer 向けにカスタマイズできます。

手順

1. Web ブラウザーで Spring Initializr を開きます。

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

2. プロジェクト、言語、Spring Boot バージョンを選択し、プロジェクトのメタデータを入力します。
3. **Add Dependencies** をクリックし、**Spring Web** を選択して **spring-boot-starter-web** の依存関係を追加します。
4. **GENERATE** をクリックします。プロジェクトの ZIP ファイルがダウンロードされます。
5. ZIP ファイルを展開して、Spring Boot プロジェクトディレクトリーに移動します。
6. Red Hat Business Optimizer の依存関係 (**optaplanner-spring-boot-starter**) を追加します。現在、**optaplanner-spring-boot-starter** は Spring Initializr に含まれていないため、ビルドファイルに手動で追加する必要があります。
 - Maven プロジェクトを生成した場合は **optaplanner-spring-boot-starter** 依存関係をプロジェクトの **pom.xml** に追加します。

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

以下の例では、**optaplanner-spring-boot-starter** 依存関係をプロジェクトの **pom.xml** ファイルに追加します。

```
<?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>
```

- Gradle プロジェクトを生成した場合は **optaplanner-spring-boot-starter** を **build.gradle** ファイルに追加します。

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

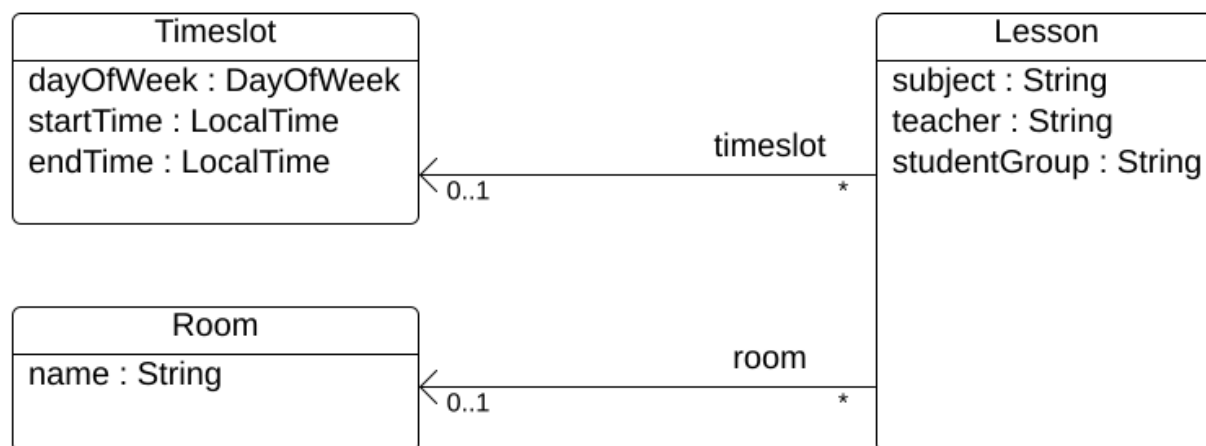
以下の例では **optaplanner-spring-boot-starter** 依存関係を、Gradle Java プロジェクトの **build.gradle** ファイルに追加します。

```
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()  
}
```

第2章 ドメインオブジェクトのモデル化

Red Hat Business Optimizer の時間割プロジェクトの目標は、レッスンごとに時間枠と部屋に割り当てることです。これには、次の図に示すように、**Timeslot**、**Lesson**、および **Room** の3つのクラスを追加します。

Time table class diagram



Timeslot

Timeslot クラスは、**Monday 10:30 - 11:30**、**Tuesday 13:30 - 14:30** など、授業の長さを表します。この例では、時間枠はすべて同じ長さ (期間) で、昼休みまたは他の休憩時間にはこのスロットはありません。

高校のスケジュールは毎週 (同じ内容が) 繰り返されるだけなので、時間枠には日付がありません。また、[継続的プランニング](#) は必要ありません。解決時に **Timeslot** インスタンスが変更しないため、**Timeslot** は **問題ファクト** と呼ばれます。このようなクラスには Red Hat Business Optimizer 固有のアノテーションは必要ありません。

Room

Room クラスは、**Room A**、**Room B** など、授業の場所を表します。以下の例では、どの部屋も定員制限がなく、すべての授業に対応できます。

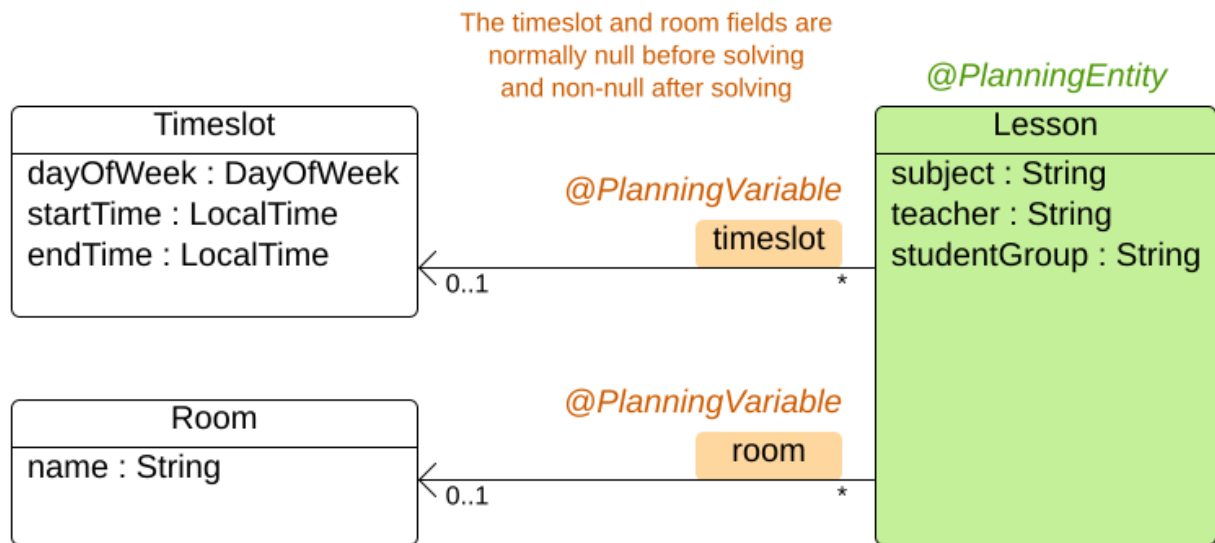
Room インスタンスは解決時に変化しないため、**Room** は **問題ファクト** でもあります。

Lesson

授業中 (**Lesson** クラスで表現)、教師は複数の生徒に **Math by A.Turing for 9th grade**、**Chemistry by M.Curie for 10th grade** などの教科を指導します。ある教科について、毎週複数回、同じ教師が同じ生徒グループを指導する場合は、**Lesson** インスタンスが複数使用されますが、それらは **id** で識別可能です。たとえば、9 年生の場合は、1 週間に 6 回数学の授業があります。

解決中に、Red Hat Business Optimizer は、**Lesson** クラスの **timeslot** フィールドと **room** フィールドを変更して、各授業を、時間枠 1 つ、部屋 1 つに割り当てます。Red Hat Business Optimizer はこれらのフィールドを変更するため、**Lesson** は **プランニングエンティティ** となります。

Time table class diagram



前図では、オレンジのフィールド以外のほぼすべてのフィールドに、入力データが含まれています。授業の **timeslot** フィールドと **room** フィールドは、入力データに割り当てられておらず (**null**)、出力データに割り当てられて (**null** ではない) います。Red Hat Business Optimizer は、解決時にこれらのフィールドを変更します。このようなフィールドはプランニング変数と呼ばれます。このフィールドを Red Hat Business Optimizer に認識させるには、**timeslot** フィールドと **room** のフィールドに **@PlanningVariable** アノテーションが必要です。このフィールドに含まれる **Lesson** クラスには、**@PlanningEntity** アノテーションが必要です。

手順

1. **src/main/java/com/example/domain/Timeslot.java** クラスを作成します。

```

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;
    }
}

```

後述しているように、**toString()** メソッドで出力を短くするため、Red Hat Business Optimizer の **DEBUG** ログまたは **TRACE** ログの読み取りが簡単になっています。

2. **src/main/java/com/example/domain/Room.java** クラスを作成します。

```

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. **src/main/java/com/example/domain/Lesson.java** クラスを作成します。

```
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;
    }
}
```

Lesson クラスには **@PlanningEntity** アノテーションが含まれており、その中にプランニング変数が1つ以上含まれているため、Red Hat Business Optimizer はこのクラスが解決時に変化することを認識します。

timeslot フィールドには **@PlanningVariable** アノテーションがあるため、Red Hat Business Optimizer は、このフィールドの値が変化することを認識しています。このフィールドに割り当てることのできる **Timeslot** インスタンスを見つけ出すために、Red Hat Business Optimizer は **valueRangeProviderRefs** プロパティを使用して値の範囲プロバイダーと連携し、**List<Timeslot>** を提供して選択できるようにします。値の範囲プロバイダーに関する詳細は、[4章 プランニングソリューションでのドメインオブジェクトの収集](#) を参照してください。

room フィールドにも、同じ理由で **@PlanningVariable** アノテーションが含まれます。

第3章 制約の定義およびスコアの計算

問題の解決時に **スコア** で導かれた解の質を表します。スコアが高いほど質が高くなります。Red Hat Business Optimizer は、利用可能な時間内で見つかった解の中から最高スコアのものを探し出します。これが **最適** 解である可能性があります。

時間割の例のユースケースでは、ハードとソフト制約を使用しているため、**HardSoftScore** クラスでスコアを表します。

- ハード制約は、絶対に違反しないでください。たとえば、**部屋に同時に割り当てることができる授業は、最大1コマです。**
- ソフト制約は、違反しないようにしてください。たとえば、**教師は、1つの部屋での授業を希望します。**

ハード制約は、他のハード制約と比べて、重み付けを行います。ソフト制約は、他のソフト制約と比べて、重み付けを行います。ハード制約は、それぞれの重みに関係なく、常にソフト制約よりも高くなります。

EasyScoreCalculator クラスを実装して、スコアを計算できます。

```
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);
    }
}
```

残念ながら、この解は漸増的ではないので、適切にスケーリングされません。授業が別の時間枠や教室に割り当てられるたびに、全授業が再評価され、新しいスコアが計算されます。

`src/main/java/com/example/solver/TimeTableConstraintProvider.java` クラスを作成して、漸増的スコア計算を実行すると、解がより優れたものになります。このクラスは、Java 8 Streams と SQL を基にした Red Hat Business Optimizer の `ConstraintStream` API を使用します。**ConstraintProvider** は、**EasyScoreCalculator** と比べ、スケーリングの規模が遥かに大きくなっています ($O(n^2)$ ではなく $O(n)$)。

手順

以下の `src/main/java/com/example/solver/TimeTableConstraintProvider.java` クラスを作成します。

```
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);
    }
}
```

第4章 プランニングソリューションでのドメインオブジェクトの収集

TimeTable インスタンスは、単一データセットの **Timeslot** インスタンス、**Room** インスタンス、および **Lesson** インスタンスをラップします。さらに、このインスタンスは、特定のプランニング変数の状態を持つ授業がすべて含まれているため、このインスタンスは **プランニングソリューション** となり、スコアが割り当てられます。

- 授業がまだ割り当てられていない場合は、スコアが **-4init/0hard/0soft** のソリューションなど、**初期化されていない** ソリューションとなります。
- ハード制約に違反する場合、スコアが **-2hard/-3soft** のソリューションなど、**実行不可** なソリューションとなります。
- 全ハード制約に準拠している場合は、スコアが **0hard/-7soft** など、**実行可能** なソリューションとなります。

TimeTable クラスには **@PlanningSolution** アノテーションが含まれているため、Red Hat Business Optimizer はこのクラスに全入出力データが含まれていることを認識します。

具体的には、このクラスは問題の入力です。

- 全時間枠が含まれる **timeslotList** フィールド
 - これは、解決時に変更されないため、問題ファクトリーストです。
- 全部屋が含まれる **roomList** フィールド
 - これは、解決時に変更されないため、問題ファクトリーストです。
- 全授業が含まれる **lessonList** フィールド
 - これは、解決時に変更されるため、プランニングエンティティです。
 - 各 **Lesson**:
 - **timeslot** フィールドおよび **room** フィールドの値は通常、**null** で未割り当てです。これらの値は、プランニング変数です。
 - **subject**、**teacher**、**studentGroup** などの他のフィールドは入力されます。これらのフィールドは問題プロパティです。

ただし、このクラスはソリューションの出力でもあります。

- **Lesson** インスタンスごとの **lessonList** フィールドには、解決後は **null** ではない **timeslot** フィールドと **room** フィールドが含まれます。
- 出力ソリューションの品質を表す **score** フィールド (例: **0hard/-5soft**)

手順

src/main/java/com/example/domain/TimeTable.java クラスを作成します。

```
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;
    }

}

```

値の範囲のプロバイダー

timeslotList フィールドは、値の範囲プロバイダーです。これは **Timeslot** インスタンスを保持し、Red Hat Business Optimizer がこのインスタンスを選択して、**Lesson** インスタンスの **timeslot** フィールドに割り当てることができます。**timeslotList** フィールドには **@ValueRangeProvider** アノテーションがあり、**id** を、**Lesson** の **@PlanningVariable** の **valueRangeProviderRefs** に一致させます。

同じロジックに従い、**roomList** フィールドにも **@ValueRangeProvider** アノテーションが含まれています。

問題ファクトとプランニングエンティティのプロパティ

さらに Red Hat Business Optimizer は、変更可能な **Lesson** インスタンス、さらに **TimeTableConstraintProvider** によるスコア計算に使用する **Timeslot** インスタンスと **Room** インスタンスを取得する方法を把握しておく必要があります。

timeslotList フィールドと **roomList** フィールドには **@ProblemFactCollectionProperty** アノテーションが含まれているため、**TimeTableConstraintProvider** はこれらのインスタンスから選択できます。

lessonList には **@PlanningEntityCollectionProperty** アノテーションが含まれているため、Red Hat Business Optimizer は解決時に変更でき、**TimeTableConstraintProvider** はこの中から選択できます。

第5章 TIMETABLE サービスの作成

これですべて組み合わせ、REST サービスを作成する準備ができました。しかし、REST スレッドで計画問題を解決すると、HTTP タイムアウトの問題が発生します。そのため、Spring Boot スターターでは **SolverManager** を注入することで、個別のスレッドプールでソルバーを実行して複数のデータセットを並行して解決できます。

手順

`src/main/java/com/example/solver/TimeTableController.java` クラスを作成します。

```
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;
    }
}
```

この例では、初期実装はソルバーが完了するのを待つので、HTTP タイムアウトがまだ発生します。**complete** 実装を使用することで、より適切に HTTP タイムアウトを回避できます。

第6章 ソルバー終了時間の設定

プランニングアプリケーションに終了設定または終了イベントがない場合、理論的には永続的に実行されることになり、実際には HTTP タイムアウトエラーが発生します。これが発生しないようにするには、**optaplanner.solver.termination.spent-limit** パラメーターを使用して、アプリケーションが終了してから時間を指定します。多くのアプリケーションでは、この時間を最低でも 5 分 (**5m**) に設定します。ただし、時間割の例では、解決時間を 5 分に制限すると、期間が十分に短いため HTTP タイムアウトを回避できます。

手順

以下の内容を含む **src/main/resources/application.properties** ファイルを作成します。

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


第7章 アプリケーションを実行可能にする手順

Red Hat Business Optimizer Spring Boot の時間割プロジェクトを完了すると、標準 Java **main()** メソッドで駆動する1つの実行可能 JAR ファイルにすべてをパッケージ化します。

前提条件

- これで Red Hat Business Optimizer Spring Boot の時間割プロジェクトが完成しました。

手順

1. 以下の内容を含む **TimeTableSpringBootApplication.java** クラスを作成します。

```
package com.example;

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

@SpringBootApplication
public class TimeTableSpringBootApplication {

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

}
```

2. Spring Initializr で作成された **src/main/java/com/example/DemoApplication.java** クラスは **TimeTableSpringBootApplication.java** クラスに置き換えます。
3. 通常の Java アプリケーションのメインクラスとして **TimeTableSpringBootApplication.java** クラスを実行します。

7.1. 時間割アプリケーションの試行

Red Hat Business Optimizer Spring Boot 時間割アプリケーションの起動後に、任意の REST クライアントで REST サービスをテストできます。この例では Linux **curl** コマンドを使用して POST 要求を送信します。

前提条件

- Red Hat Business Optimizer Spring Boot アプリケーションが実行中である。

手順

以下のコマンドを入力します。

```
$ 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"}]}'
```

約 5 秒後 (**application.properties** で定義された終了時間) に、サービスにより、以下の例のような出力が返されます。

```
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"}
```

アプリケーションにより、4 つの授業がすべて 2 つの時間枠、そして 2 つある部屋のうちの 1 つに割り当てられている点に注目してください。また、すべてのハード制約に準拠することに注意してください。たとえば、M. Curie の 2 つの授業は異なる時間スロットにあります。

サーバー側で **info** ログに、この 5 秒間で Red Hat Business Optimizer が行った内容が記録されます。

```
... 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. アプリケーションのテスト

適切なアプリケーションにはテストが含まれます。以下の例では、Red Hat Business Optimizer Spring Boot 時間割アプリケーションをテストします。このアプリケーションは、JUnit テストを使用してテストのデータセットを生成し、**TimeTableController** に送信して解決します。

手順

以下の内容を含む **src/test/java/com/example/solver/TimeTableControllerTest.java** クラスを作成します。

```
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);
    }
}

```

このテストは、解決後にすべての授業がタイムスロットと部屋に割り当てられていることを確認します。また、実行可能解（ハード制約の違反なし）も確認します。

通常、ソルバーは 200 ミリ秒未満で実行可能解を検索します。**@SpringBootTest** アノテーションの **properties** が、実行可能なソリューション (**0hard/*soft**) が見つかると同時に、ソルバーの終了を上書きします。こうすることで、ユニットテストが任意のハードウェアで実行される可能性があるため、ソルバーの時間をハードコード化するのを回避します。このアプローチを使用することで、動きが遅いマシンであっても、実行可能なソリューションを検索するのに十分な時間だけテストが実行されます。ただし、高速マシンでも、厳密に必要とされる時間よりもミリ秒単位で長く実行されることはありません。

7.3. ロギング

Red Hat Business Optimizer Spring Boot の時間割アプリケーションアプリケーションを完了後にロギング情報を使用すると、**ConstraintProvider** で制約が微調整しやすくなります。**info** ログファイルでスコア計算の速度を確認して、制約に加えた変更の影響を評価します。デバッグモードでアプリケーションを実行して、アプリケーションが行う手順をすべて表示するか、追跡ログを使用して全手順および動きをロギングします。

手順

1. 時間割アプリケーションを一定の時間 (例: 5 分) 実行します。
2. 以下の例のように、**log** ファイルのスコア計算の速度を確認します。

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

3. 制約を変更して、同じ時間、プランニングアプリケーションを実行し、**log** ファイルに記録されているスコア計算速度を確認します。
4. アプリケーションをデバッグモードで実行して、全手順をログに記録します。
 - コマンドラインからデバッグモードを実行するには、**-D** システムプロパティを使用します。
 - **application.properties** ファイルのロギングを変更するには、以下の行をこのファイルに追加します。

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

以下の例では、デバッグモードでの **log** ファイルの出力を表示します。

```

... 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. **trace** ロギングを使用して、全手順、および手順ごとの全動きを表示します。

第8章 データベースと UI 統合の追加

Spring Boot を使用して Red Hat Business Optimizer アプリケーションの例を作成したら、データベースと UI 統合を追加します。

前提条件/事前作業

- Red Hat Business Optimizer Spring Boot の時間割サンプルが作成されている。

手順

1. **Timeslot**、**Room**、および **Lesson** の Java Persistence API (JPA) リポジトリを作成します。JPA リポジトリの作成に関する情報は、Spring の Web サイトの [Accessing Data with JPA](#) を参照してください。
2. REST で JPA リポジトリを公開します。リポジトリの公開に関する情報は、Spring の Web サイトの [Accessing JPA Data with REST](#) を参照してください。
3. **TimeTableRepository** Facade をビルドして、1 回のトランザクションで **TimeTable** を読み取り、書き込みます。
4. 以下の例のように **TimeTableController** を調整します。

```
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);
    }
}

```

簡素化するために、このコードは **TimeTable** インスタンスを1つだけ処理しますが、マルチテナントを有効にして、異なる高校の **TimeTable** インスタンスを複数、平行して処理することも簡単にできます。

getTimeTable() メソッドは、データベースから最新の時間割を返します。このメソッドは、**ScoreManager** (自動注入) を使用して、UI でスコアを表示できるように、対象の時間割のスコアを計算します。

solve() メソッドは、ジョブを開始して、現在の時間割を解決し、時間枠と部屋の割り当てをデータベースに保存します。このメソッドは、**SolverManager.solveAndListen()** メソッドを使用して、中間の最適解をリッスンし、それに合わせてデータベースを更新します。こうすることで、バックエンドで解決しながら、UI で進捗を表示できます。

5. **solve()** メソッドが即座に返されるようになったので、以下の例のように **TimeTableControllerTest** を調整します。

```

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. ソルバーの解決が完了するまで、最新のソリューションをポーリングします。
7. 時間割を視覚化するには、これらの REST メソッドの上に適切な Web UI を構築します。

付録A バージョン情報

本書の最終更新日: 2022 年 3 月 8 日 (火)