OptaPlanner - Using AI to optimize a schedule with OptaPlanner

This guide walks you through the process of creating a Quarkus application with OptaPlanner's constraint solving Artificial Intelligence (AI).

What you will build

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 hard and soft scheduling *constraints*, such as:

- 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.

- A student dislikes sequential lessons on the same subject.

Mathematically speaking, school timetabling is an *NP-hard* problem. That means it is difficult to scale.

Simply brute force iterating through all possible combinations takes millions of years for a non-trivial dataset, even on a supercomputer. Luckily, AI constraint solvers such as OptaPlanner have advanced algorithms that deliver a near-optimal solution in a reasonable amount of time.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `optaplanner-quickstart` directory.

Prerequisites

To complete this guide, you need:

- about 30 minutes

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.6.3 or Gradle 4+

The build file and the dependencies

Use code.quarkus.io to generate an application with the following extensions, for Maven or Gradle:

- RESTEasy JAX-RS (`quarkus-resteasy`)

- RESTEasy Jackson (`quarkus-resteasy-jackson`)

- OptaPlanner (`optaplanner-quarkus`)

- OptaPlanner Jackson (`optaplanner-quarkus-jackson`)

Alternatively, generate it from the command line with Maven:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.2.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=optaplanner-quickstart \
    -Dextensions="resteasy, resteasy-jackson, optaplanner-quarkus,
optaplanner-quarkus-jackson"
cd optaplanner-quickstart
```

In Maven, your `pom.xml` file contains these dependencies:

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-universe-bom</artifactId>
      <version>1.7.2.Final</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy</artifactId>
  </dependency>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jackson</artifactId>
  </dependency>
  <dependency>
    <groupId>org.optaplanner</groupId>
    <artifactId>optaplanner-quarkus</artifactId>
  </dependency>
  <dependency>
    <groupId>org.optaplanner</groupId>
    <artifactId>optaplanner-quarkus-jackson</artifactId>
  </dependency>

  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```
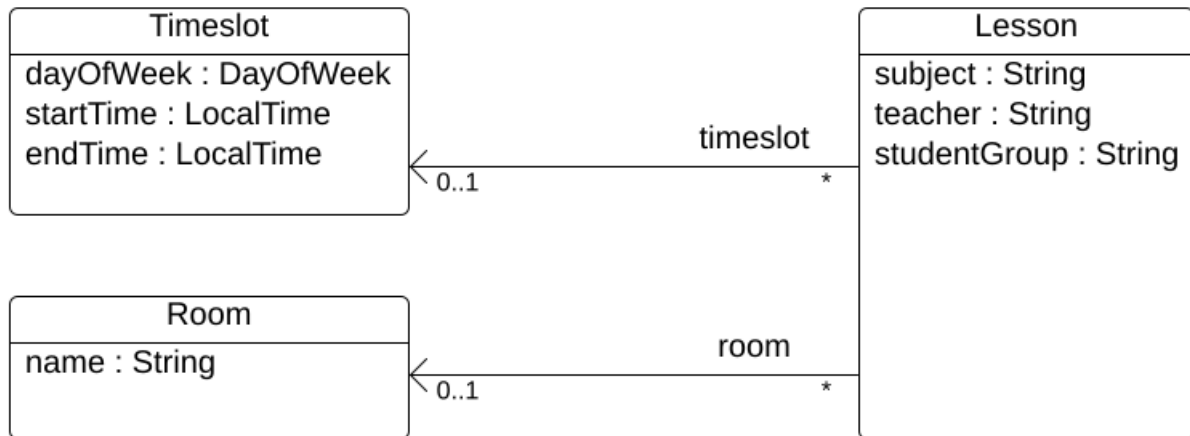
Model the domain objects

Your goal is to assign each lesson to a time slot and a room. You will create these classes:

Time table class 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`. For simplicity's sake, 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. So there is no need for continuous planning.

Create the `src/main/java/org/acme/optaplanner/domain/Timeslot.java` class:

```java
package org.acme.optaplanner.domain;

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

public class Timeslot {

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

    public Timeslot() {
    }

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

    public DayOfWeek getDayOfWeek() {
        return dayOfWeek;
    }

    public LocalTime getStartTime() {
        return startTime;
    }

    public LocalTime getEndTime() {
        return endTime;
    }

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

}
```

Because no `Timeslot` instances change during solving, a `Timeslot` is called a *problem fact*. Such classes do not require any OptaPlanner specific annotations.

Notice the `toString()` method keeps the output short, so it is easier to read OptaPlanner's `DEBUG` or `TRACE` log, as shown later.

Room

The Room class represents a location where lessons are taught, for example, Room A or Room B. For simplicity's sake, all rooms are without capacity limits and they can accommodate all lessons.

Create the src/main/java/org/acme/optaplanner/domain/Room.java class:

```java
package org.acme.optaplanner.domain;

public class Room {

    private String name;

    public Room() {
    }

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

    public String getName() {
        return name;
    }

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

}
```

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 per 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, OptaPlanner changes the timeslot and room fields of the Lesson class, to assign each lesson to a time slot and a room. Because OptaPlanner changes these fields, Lesson is a *planning entity*:

Time table class diagram

The timeslot and room fields are normally null before solving and non-null after solving

@PlanningEntity

Timeslot
dayOfWeek : DayOfWeek
startTime : LocalTime
endTime : LocalTime

@PlanningVariable
timeslot

0..1   *

Lesson
subject : String
teacher : String
studentGroup : String

Room
name : String

@PlanningVariable
room

0..1   *
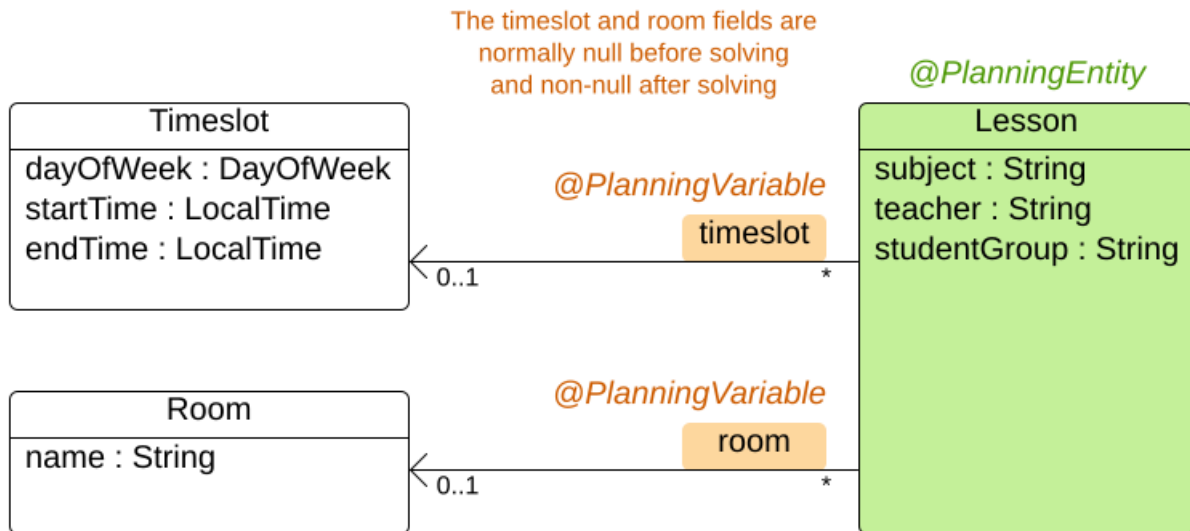
Most of the fields in the previous diagram contain input data, except for the orange fields: A lesson's `timeslot` and `room` fields are unassigned (`null`) in the input data and assigned (not `null`) in the output data. OptaPlanner changes these fields during solving. Such fields are called planning variables. In order for OptaPlanner to recognize them, both the `timeslot` and `room` fields require an `@PlanningVariable` annotation. Their containing class, `Lesson`, requires an `@PlanningEntity` annotation.

Create the `src/main/java/org/acme/optaplanner/domain/Lesson.java` class:

```java
package org.acme.optaplanner.domain;

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

@PlanningEntity
public class Lesson {

    @PlanningId
    private Long id;

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

    @PlanningVariable(valueRangeProviderRefs = "timeslotRange")
    private Timeslot timeslot;
    @PlanningVariable(valueRangeProviderRefs = "roomRange")
    private Room room;
```

```java
    public Lesson() {
    }

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

    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;
    }

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

}
```

The `Lesson` class has an `@PlanningEntity` annotation, so OptaPlanner knows that this class changes during solving because it contains one or more planning variables.

The `timeslot` field has an `@PlanningVariable` annotation, so OptaPlanner knows that it can change its value. In order to find potential `Timeslot` instances to assign to this field, OptaPlanner uses the `valueRangeProviderRefs` property to connect to a value range provider (explained later) that provides a `List<Timeslot>` to pick from.

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

> **ℹ** Determining the `@PlanningVariable` fields for an arbitrary constraint solving use case is often challenging the first time. Read the domain modeling guidelines to avoid common pitfalls.

Define the constraints and calculate the score

A *score* represents the quality of a given solution. The higher the better. OptaPlanner looks for the best solution, which is the solution with the highest score found in the available time. It could be the *optimal* solution.

Because this 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 too, 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:

```java
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 optaplanner-
quickstart
        return HardSoftScore.of(hardScore, softScore);
    }

}
```

Unfortunately **that 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.

Instead, create a src/main/java/org/acme/optaplanner/solver/TimeTableConstraintProvider.java class to perform incremental score calculation. It uses OptaPlanner's ConstraintStream API which is inspired by Java 8 Streams and SQL:

```java
package org.acme.optaplanner.solver;

import org.acme.optaplanner.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
optaplanner-quickstart
        };
    }

    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
                .fromUniquePair(Lesson.class,
                        Joiners.equal(Lesson::getTimeslot),
                        Joiners.equal(Lesson::getTeacher))
                .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
                .fromUniquePair(Lesson.class,
                        Joiners.equal(Lesson::getTimeslot),
                        Joiners.equal(Lesson::getStudentGroup))
                .penalize("Student group conflict",
HardSoftScore.ONE_HARD);
    }

}
```

The `ConstraintProvider` scales an order of magnitude better than the `EasyScoreCalculator`: $O(n)$ instead of $O(n^2)$.

Gather the domain objects in a planning solution

A `TimeTable` 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`.

Create the `src/main/java/org/acme/optaplanner/domain/TimeTable.java` class:

```
package org.acme.optaplanner.domain;

import java.util.List;

import
```

```java
org.optaplanner.core.api.domain.solution.PlanningEntityCollectionPr
operty;
import org.optaplanner.core.api.domain.solution.PlanningScore;
import org.optaplanner.core.api.domain.solution.PlanningSolution;
import
org.optaplanner.core.api.domain.solution.drools.ProblemFactCollecti
onProperty;
import
org.optaplanner.core.api.domain.valuerange.ValueRangeProvider;
import
org.optaplanner.core.api.score.buildin.hardsoft.HardSoftScore;

@PlanningSolution
public class TimeTable {

    @ProblemFactCollectionProperty
    @ValueRangeProvider(id = "timeslotRange")
    private List<Timeslot> timeslotList;
    @ProblemFactCollectionProperty
    @ValueRangeProvider(id = "roomRange")
    private List<Room> roomList;
    @PlanningEntityCollectionProperty
    private List<Lesson> lessonList;

    @PlanningScore
    private HardSoftScore score;

    public TimeTable() {
    }

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

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

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

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

    public HardSoftScore getScore() {
```

```
        return score;
    }

}
```

The `TimeTable` class has an `@PlanningSolution` annotation, so OptaPlanner 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`

The value range providers

That `timeslotList` field is a value range provider. It holds the `Timeslot` instances which OptaPlanner 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, OptaPlanner 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 OptaPlanner can change them during solving and your `TimeTableConstraintProvider` can select *from* those too.

Create the solver 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 Quarkus extension injects a `SolverManager`, which runs solvers in a separate thread pool and can solve multiple datasets in parallel.

Create the `src/main/java/org/acme/optaplanner/rest/TimeTableResource.java` class:

```java
package org.acme.optaplanner.rest;

import java.util.UUID;
import java.util.concurrent.ExecutionException;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.acme.optaplanner.domain.TimeTable;
import org.optaplanner.core.api.solver.SolverJob;
import org.optaplanner.core.api.solver.SolverManager;

@Path("/timeTable")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class TimeTableResource {

    @Inject
    SolverManager<TimeTable, UUID> solverManager;

    @POST
    @Path("/solve")
    public TimeTable solve(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;
    }

}
```

For simplicity's sake, this initial implementation waits for the solver to finish, which can still cause an HTTP timeout. The *complete* implementation avoids HTTP timeouts much more elegantly.

Set the termination time

Without a termination setting or a termination event, the solver runs forever. To avoid that, limit the solving time to five seconds. That is short enough to avoid the HTTP timeout.

Create the src/main/resources/application.properties file:

```
# The solver runs only for 5 seconds to avoid a HTTP timeout in
this simple implementation.
# It's recommended to run for at least 5 minutes ("5m") otherwise.
quarkus.optaplanner.solver.termination.spent-limit=5s
```

Make the application executable

First start the application:

```
$ ./mvnw compile quarkus:dev
```

Try the application

Now that the application is running, you can test the REST service. You can use any REST client you wish. The following example uses the Linux command curl to send a POST request:

```
$ curl -i -X POST http://localhost:8080/timeTable/solve -H
"Content-Type:application/json" -d
'{"timeslotList":[{"dayOfWeek":"MONDAY","startTime":"08:30:00","end
Time":"09:30:00"},{"dayOfWeek":"MONDAY","startTime":"09:30:00","end
Time":"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, according to the termination spent time defined in your 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","end
Time":"09:30:00"},"room":{"name":"Room
A"}},{"id":2,"subject":"Chemistry","teacher":"M.
Curie","studentGroup":"9th
grade","timeslot":{"dayOfWeek":"MONDAY","startTime":"09:30:00","end
Time":"10:30:00"},"room":{"name":"Room
A"}},{"id":3,"subject":"French","teacher":"M.
Curie","studentGroup":"10th
grade","timeslot":{"dayOfWeek":"MONDAY","startTime":"08:30:00","end
Time":"09:30:00"},"room":{"name":"Room
B"}},{"id":4,"subject":"History","teacher":"I.
Jones","studentGroup":"10th
grade","timeslot":{"dayOfWeek":"MONDAY","startTime":"09:30:00","end
Time":"10:30:00"},"room":{"name":"Room B"}}],"score":"0hard/0soft"}
```

Notice that your 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 show what OptaPlanner 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).
```

Test the application

A good application includes test coverage. In a JUnit test, generate a test dataset and send it to the `TimeTableResource` to solve.

Create the `src/test/java/org/acme/optaplanner/rest/TimeTableResourceTest.java`

class:

```java
package org.acme.optaplanner.rest;

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

import javax.inject.Inject;

import io.quarkus.test.junit.QuarkusTest;
import org.acme.optaplanner.domain.Room;
import org.acme.optaplanner.domain.Timeslot;
import org.acme.optaplanner.domain.Lesson;
import org.acme.optaplanner.domain.TimeTable;
import org.acme.optaplanner.rest.TimeTableResource;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;

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

@QuarkusTest
public class TimeTableResourceTest {

    @Inject
    TimeTableResource timeTableResource;

    @Test
    @Timeout(600_000)
    public void solve() {
        TimeTable problem = generateProblem();
        TimeTable solution = timeTableResource.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)));
```

```java
        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).

Add test properties to the src/main/resources/application.properties file:

```
# The solver runs only for 5 seconds to avoid a HTTP timeout in
this simple implementation.
# It's recommended to run for at least 5 minutes ("5m") otherwise.
quarkus.optaplanner.solver.termination.spent-limit=5s

# Effectively disable this termination in favor of the best-score-
limit
%test.quarkus.optaplanner.solver.termination.spent-limit=1h
%test.quarkus.optaplanner.solver.termination.best-score-
limit=0hard/*soft
```

Normally, the solver finds a feasible solution in less than 200 milliseconds. Notice how the `application.properties` overwrites the solver termination during tests 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. But it does not run a millisecond longer than it strictly must, even on fast machines.

Logging

When adding constraints in your `ConstraintProvider`, keep an eye on the *score calculation speed* in the `info` log, after solving for the same amount of time, to assess the performance impact:

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

To understand how OptaPlanner is solving your problem internally, change the logging in the `application.properties` file or with a `-D` system property:

```
quarkus.log.category."org.optaplanner".level=debug
```

Use debug logging to show every *step*:

```
... 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}]).
...
```

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

Summary

Congratulations! You have just developed a Quarkus application with OptaPlanner!

Further improvements: Database and UI integration

Now try adding database and UI integration:

1. Store `Timeslot`, `Room`, and `Lesson` in the database with Hibernate and Panache.

2. Expose them through REST.

3. Adjust the `TimeTableResource` to read and write a `TimeTable` in a single transaction and use those accordingly:

```java
package org.acme.optaplanner.rest;

import javax.inject.Inject;
import javax.transaction.Transactional;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import io.quarkus.panache.common.Sort;
import org.acme.optaplanner.domain.Lesson;
import org.acme.optaplanner.domain.Room;
import org.acme.optaplanner.domain.TimeTable;
import org.acme.optaplanner.domain.Timeslot;
import org.optaplanner.core.api.score.ScoreManager;
import org.optaplanner.core.api.solver.SolverManager;
import org.optaplanner.core.api.solver.SolverStatus;

@Path("/timeTable")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class TimeTableResource {

    public static final Long SINGLETON_TIME_TABLE_ID = 1L;

    @Inject
    SolverManager<TimeTable, Long> solverManager;
    @Inject
    ScoreManager<TimeTable> scoreManager;
```

```java
    // To try, open http://localhost:8080/timeTable
    @GET
    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 = findById(SINGLETON_TIME_TABLE_ID);
        scoreManager.updateScore(solution); // Sets the score
        solution.setSolverStatus(solverStatus);
        return solution;
    }

    @POST
    @Path("/solve")
    public void solve() {
        solverManager.solveAndListen(SINGLETON_TIME_TABLE_ID,
                this::findById,
                this::save);
    }

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

    @POST
    @Path("/stopSolving")
    public void stopSolving() {
        solverManager.terminateEarly(SINGLETON_TIME_TABLE_ID);
    }

    @Transactional
    protected TimeTable findById(Long id) {
        if (!SINGLETON_TIME_TABLE_ID.equals(id)) {
            throw new IllegalStateException("There is no
timeTable with id (" + id + ").");
        }
        // Occurs in a single transaction, so each initialized
lesson references the same timeslot/room instance
        // that is contained by the timeTable's
timeslotList/roomList.
        return new TimeTable(

Timeslot.listAll(Sort.by("dayOfWeek").and("startTime").and("endT
ime").and("id")),
                Room.listAll(Sort.by("name").and("id")),

Lesson.listAll(Sort.by("subject").and("teacher").and("studentGro
```

```
up").and("id")));
    }

    @Transactional
    protected void save(TimeTable timeTable) {
        for (Lesson lesson : timeTable.getLessonList()) {
            // TODO this is awfully naive: optimistic locking
causes issues if called by the SolverManager
            Lesson attachedLesson =
Lesson.findById(lesson.getId());
            attachedLesson.setTimeslot(lesson.getTimeslot());
            attachedLesson.setRoom(lesson.getRoom());
        }
    }

}
```

For simplicity's sake, this code handles only one `TimeTable`, 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 time table from the database. It uses the `ScoreManager` (which is automatically injected) to calculate the score of that time table, so the UI can show the score.

The `solve()` method starts a job to solve the current time table 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.

4. Adjust the `TimeTableResourceTest` accordingly, now that the `solve()` method returns immediately. Poll for the latest solution until the solver finishes solving:

```
package org.acme.optaplanner.rest;

import javax.inject.Inject;

import io.quarkus.test.junit.QuarkusTest;
import org.acme.optaplanner.domain.Lesson;
import org.acme.optaplanner.domain.TimeTable;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import org.optaplanner.core.api.solver.SolverStatus;

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

@QuarkusTest
public class TimeTableResourceTest {

    @Inject
    TimeTableResource timeTableResource;

    @Test
    @Timeout(600_000)
    public void solveDemoDataUntilFeasible() throws
InterruptedException {
        timeTableResource.solve();
        TimeTable timeTable = timeTableResource.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 = timeTableResource.getTimeTable();
        }
        assertFalse(timeTable.getLessonList().isEmpty());
        for (Lesson lesson : timeTable.getLessonList()) {
            assertNotNull(lesson.getTimeslot());
            assertNotNull(lesson.getRoom());
        }
        assertTrue(timeTable.getScore().isFeasible());
    }

}
```

5. Build an attractive web UI on top of these REST methods to visualize the timetable.

Take a look at the solution to see how this all turns out.