OptaWeb Employee Rostering User Guide

The OptaPlanner Team

Version 8.23.0-SNAPSHOT

Table of Contents

Chapter 1. OptaWeb Employee Rostering Introduction

1.1. What is OptaWeb Employee Rostering?

Every organization faces planning problems: providing products or services with a limited set of *constrained* resources (employees, assets, time and money). One such planning problem is employee shift rostering: assigning shifts to employees. OptaWeb is a web application and REST service that solves employee shift rostering problems using the OptaPlanner engine.

1.2. Build and Run the Application

To build the project with Maven, run the following command in the project's root directory:

```
mvn clean install -DskipTests
```

After building the project, run the application with:

```
java -jar optaweb-employee-rostering-standalone/target/optaweb-employee-rostering-
standalone-*-exec.jar
```

Then open http://localhost:8080/ to see the web application.

Alternatively, run `npm start` in the `optaweb-employee-rostering-frontend` directory to start the frontend in one terminal, and run `mvn quarkus:dev` in the `optaweb-employee-rostering-backend` directory to start the backend in another terminal.

To run on another port, use `-Dquarkus.http.port=⋯`:

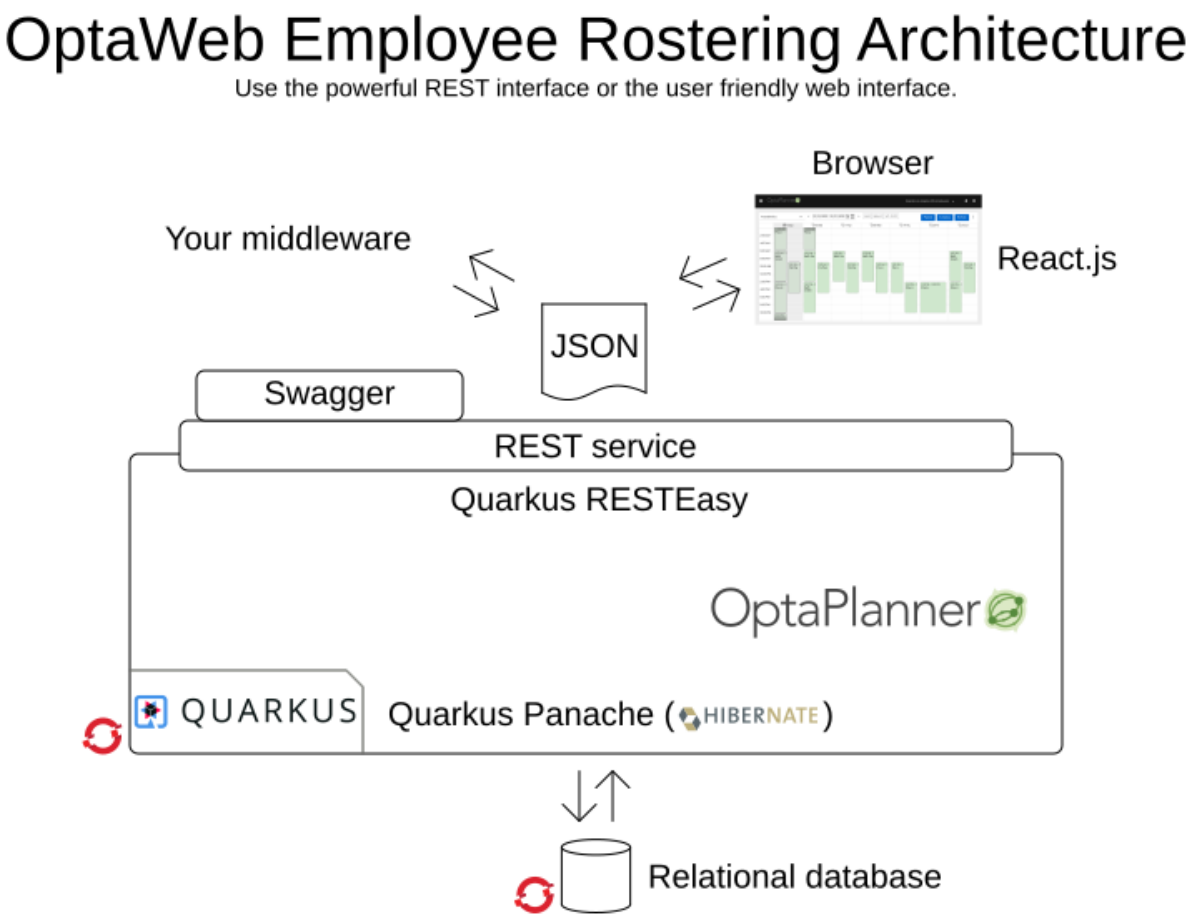
```
java -Dquarkus.http.port=18080 -jar optaweb-employee-rostering-
standalone/target/quarkus-app/quarkus-run.jar
```

1.3. System Properties

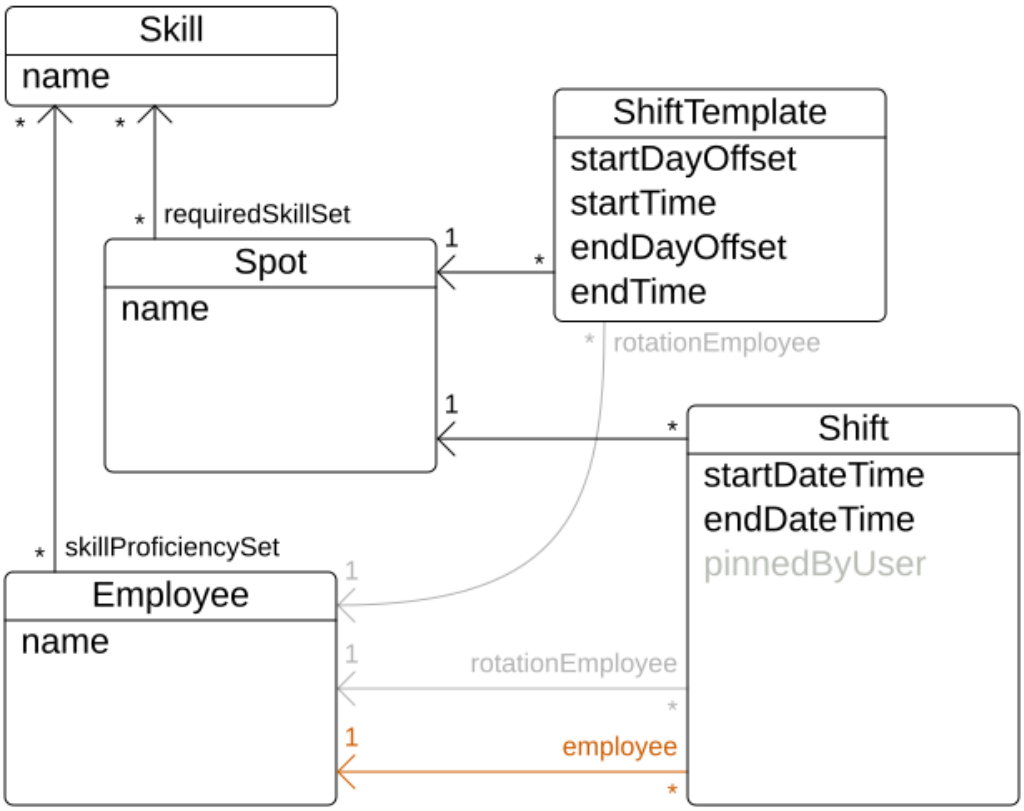
These system properties can overwrite default properties of the application, for example, by passing `-Doptaweb.generator.zoneId="America/New_York"` to Quarkus. These system properties might also be exposed as OpenShift template parameters.

- **optaweb.generator.timeZoneId**: The time zone ID for the automatically generated tenants. For example `America/New_York`. This defaults to the system default Zone ID.
- **optaweb.generator.initial.data**: What data to initially put in the database. Supported values are: `EMPTY` (no data) and `DEMO_DATA` (several tenants of various sizes). This defaults to `DEMO_DATA`

Chapter 2. Architecture

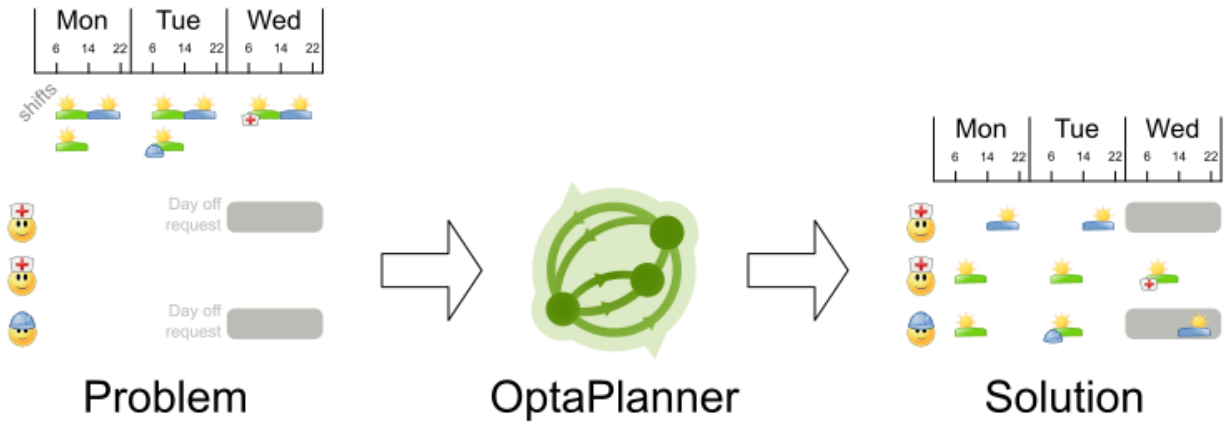


OptaWeb Employee Rostering Class Diagram



Solving with OptaPlanner

OptaPlanner automatically assigns the shifts, according to our constraints.



Chapter 3. Project Structure

The project is structured in the following folders:

- **optaweb-employee-rostering-backend** core planning domain model and backend REST api implemented with Quarkus
- **optaweb-employee-rostering-benchmark** solution benchmark
- **optaweb-employee-rostering-distribution** assembly logic for the deployment assets
- **optaweb-employee-rostering-docs** this documentation
- **optaweb-employee-rostering-frontend** user interface implemented with ReactJS

3.1. Domain Model

The domain model is the most important piece of a project based on OptaPlanner. A careful design simplifies the constraint definition. The classes of the domain model are placed in the **optaweb-employee-rostering-backend** module.

The most important classes to understand the domain model are:

- **Shift** is the *planning entity*, where is the defined the relationship with the *planning variable* employee. Other important fields: `spot`, `rotationEmployee`, `startDateTime`, `endDateTime`.
- **Employee** is the *planning variable*, it's identified by the `name` and has a set of skills (`skillProficiencySet`).
- **Roster** is the *planning solution*, `employeeList` is the list of employees (the range of values that can be assigned to the *planning variable*), the field `score` holds the score (3 levels: hard, medium, soft), the other problem facts are: `skillList`, `spotList`, `employeeAvailabilityList`, `rosterConstraintConfiguration`, `rosterState`, `shiftList`.

3.2. Constraints

The constraints are defined in the **optaweb-employee-rostering-backend** module, with the implementation of the backend REST service.

- The solver configuration file: `optaweb-employee-rostering-backend/src/main/resources/org/optaweb/employeerostering/service/solver/employeeRosteringSolverConfig.xml`
- The constraints definition file: `optaweb-employee-rostering-backend/src/main/resources/org/optaweb/employeerostering/service/solver/employeeRosteringScoreRules.drl`

3.2.1. Constraint definition

The constraints are defined using the DRL language. See: Implementing a score rule.

3.2.1.1. Hard Constraints

Required skill for a shift

```
rule "Required skill for a shift"
    when
        Shift(
                employee != null,
                !getEmployee().hasSkills(getSpot().getRequiredSkillSet()))
    then
        scoreHolder.addHardConstraintMatch(kcontext, -100);
end
```

Condition: there is a shift with an assigned employee that has NOT the skill set required by the spot.

Action: the hard score is decreased by 100 units.

Unavailable time slot for an employee

```
rule "Unavailable time slot for an employee"
    when
        EmployeeAvailability(
                state == EmployeeAvailabilityState.UNAVAILABLE,
                $e : employee,
                $startDateTime : startDateTime,
                $endDateTime : endDateTime)
        Shift(
                employee == $e,
                DateTimeUtils.doTimeslotsIntersect($startDateTime,$endDateTime,
                                                   startDateTime, endDateTime))
    then
        scoreHolder.addHardConstraintMatch(kcontext, -50);
end
```

Condition: Given an employee unavailability, there is a shift for this employee, the date time interval of the shift intersects the date time interval of the unavailability.

Action: The hard score is decreased by 50 units.

At most one shift assignment per day per employee

```
rule "At most one shift assignment per day per employee"
    when
        $s : Shift(
                employee != null,
                $e : employee,
                $leftDay : startDateTime.toLocalDate())
        Shift(
                employee == $e,
                startDateTime.toLocalDate() == $leftDay,
                this != $s)
    then
        scoreHolder.addHardConstraintMatch(kcontext, -10);
end
```

Condition: There are two shifts assigned to the same employee, the start date of one shift is equal to the start date of the other shift.

Action: The hard score is decreased by 10 units.

This rule triggers for any combination of shifts for each employee. So considering `n` employees and `m` shifts, it triggers `n*m^2` times. Luckily, the rules triggers just for shifts that are impacted by a change.

No 2 shifts within 10 hours from each other

```
rule "No 2 shifts within 10 hours from each other"
    when
        $s : Shift(
                employee != null,
                $e : employee,
                $leftEndDateTime : endDateTime)
        Shift(
                employee == $e,
                $leftEndDateTime <= endDateTime,
                $leftEndDateTime.until(startDateTime, ChronoUnit.HOURS) < 10,
                this != $s)
    then
        scoreHolder.addHardConstraintMatch(kcontext, -1);
end
```

Condition: There are two shifts assigned to the same employee, the end time of the *left* shift is prior of the other end time, the time difference between the end time of the *left* shift and the start time of the other is less than 10 hours.

Action: The hard score is decreased by 1 unit.

Daily minutes must not exceed contract maximum

```
rule "Daily minutes must not exceed contract maximum"
    when
        $employee : Employee($contract : contract, $contract.getMaximumMinutesPerDay()
!= null)
        $s : Shift(employee == $employee, $startDateTime : startDateTime)
        Number( intValue > $contract.getMaximumMinutesPerDay() ) from accumulate(
            Shift(employee == $employee, $shiftStart : startDateTime,
            $shiftEnd : endDateTime,
            $shiftStart.toLocalDate().equals($startDateTime.toLocalDate())),
            sum(Duration.between($shiftStart, $shiftEnd).toMinutes())
        )
    then
        scoreHolder.addHardConstraintMatch(kcontext, -1);
end
```

Condition: The sum of the total minutes assigned to one employee in a day is greater than the maximum minutes specified by the employee's contract.

Action: The hard score is decreased by 1 unit.

The remaining three hard constraints are similar to this last one, but for different time frames specified by the contract (weekly, monthly, yearly).

3.2.1.2. Medium Constraints

Assign every shift

```
rule "Assign every shift"
    when
        Shift(employee == null)
    then
        scoreHolder.addMediumConstraintMatch(kcontext, -1);
end
```

Condition: There is a shift with no employees assigned.

Action: The medium score is decreased by 1 unit.

3.2.1.3. Soft Constraints

Undesired time slot for an employee

```
rule "Undesired time slot for an employee"
    when
        $rosterConstraintConfiguration : RosterConstraintConfiguration
(undesiredTimeSlotWeight != 0)
        EmployeeAvailability(
                state == EmployeeAvailabilityState.UNDESIRED,
                $e : employee,
                $startDateTime : startDateTime,
                $endDateTime : endDateTime)
        Shift(
                employee == $e,
                DateTimeUtils.doTimeslotsIntersect($startDateTime,$endDateTime,
                                                   startDateTime, endDateTime))
    then
        scoreHolder.addSoftConstraintMatch(kcontext, -$rosterConstraintConfiguration
.getUndesiredTimeSlotWeight());
end
```

The first line of the `when` clause is a technique to dynamically change the weight of the constraint. If `undesiredTimeSlotWeight` is 0 the constraint is disregarded.

Condition: Given an employee's undesired date and time slot, there is a shift for this employee such that the date and time interval of the shift intersects the undesired date and time slot.

Action: The soft score is decreased by *undesiredTimeSlotWeight* units.

Desired time slot for an employee

```
rule "Desired time slot for an employee"
    when
        $rosterConstraintConfiguration : RosterConstraintConfiguration
(desiredTimeSlotWeight != 0)
        EmployeeAvailability(
                state == EmployeeAvailabilityState.DESIRED,
                $e : employee,
                $startDateTime : startDateTime,
                $endDateTime : endDateTime)
        Shift(
                employee == $e,
                DateTimeUtils.doTimeslotsIntersect($startDateTime,$endDateTime,
                                                   startDateTime, endDateTime))
    then
        scoreHolder.addSoftConstraintMatch(kcontext, +$rosterConstraintConfiguration
.getDesiredTimeSlotWeight());
end
```

The first line of the `when` clause is a technique to dynamically change the weight of the constraint. If `desiredTimeSlotWeight` is 0 the constraint is disregarded.

Condition: Given an employee desired date and time slot, there is a shift for this employee such that the date and time interval of the shift intersects the desired date and time slot.

Action: The soft score is increased by *desiredTimeSlotWeight* units.

Employee is not rotation employee

```
rule "Employee is not rotation employee"
    when
        $rosterConstraintConfiguration : RosterConstraintConfiguration
(rotationEmployeeMatchWeight != 0)
        Shift(
                rotationEmployee != null, employee != null, employee !=
rotationEmployee)
    then
        scoreHolder.addSoftConstraintMatch(kcontext, -$rosterConstraintConfiguration
.getRotationEmployeeMatchWeight());
end
```

The first line of the `when` clause is a technique to dynamically change the weight of the constraint. If `rotationEmployeeMatchWeight` is 0 the constraint is disregarded.

In general, employees desire to work following a regular schedule: a rotation plan. This represents a starting point for the actual schedule that is influenced by other factors (e.g. temporary unavailability). For this reason, all Shifts are initialized with a `rotationEmployee`.

Condition: There a shift that is assigned to an employee which is not the rotation employee.

Action: The soft score is decreased by *rotationEmployeeMatchWeight* units.

Chapter 4. Features in OptaWeb Employee Rostering

4.1. Test the JPA Database with H2

Before testing the database, make sure the application backend is running. If the application isn't running, run the following in the `optaweb-employee-rostering-backend` directory:

```
mvn quarkus:dev
```

Go to http://localhost:8080/h2-console to view the H2 database console. Enter `org.h2.Driver` in the `Driver Class` field and `jdbc:h2:mem:employeerostering` in the `JDBC URL` field, and keep the other default values. Connect, and click on the entities on the left to run SQL statements. This console allows you to view and modify the application database.

4.2. Test the REST API

As with testing the database, make sure the application backend is running to test the REST API. Go to http://localhost:8080/swagger-ui.html to view documentation and test the REST methods.