

DP1 2020-2021

Document of System Design

Tuvi's Casino Project

<https://github.com/gii-is-DP1/dp1-2020-g1-07>

Members:

- David Barragán Salazar
- Beatriz Beltrán Álvarez
- Daniel Caro Olmedo
- Antonio González Gómez
- Daniel Muñoz Heredia
- Vicente Soria Vázquez

Tutor: Bedilia Estrada

GROUP G1-07

Version 1

26/10/2020

Version history

Date	Version	General description	Sprint
06/01/2021	V1	<ul style="list-style-type: none">Document creation.	3

Contents

GROUP G1-07	1
Version 1	1
Version history.	2
Date.....	2
Introduction	5
UML Diagrams:	6
Domain Diagram	6
Layers Diagram	7
Design and architectonic patterns applied.....	8
Pattern: Proxy (Front Controller).....	8
Type: Dessign	8
Application context.....	8
Pattern: Model View Controller (MVC)	11
Type: Architectonic	11
Application context.....	11
Clases o paquetes creados.....	12
Design decisions.....	15
Decision 1: Using javascript to get the element id to complete the edit url	15
Problem description:.....	15
Justification of chosen solution.....	15
Decision 2: Tests in controllers and services	15
Problem description:.....	15
Justification of chosen solution.....	16
Decision 3: Managing the heritage relation from “Employee” entity.	16
Problem description:.....	16
Justification of chosen solution.....	17
Decision 4: Validation subsequent to @Valid in	17
Problem description:.....	17
Justification of chosen solution.....	18
Decision 5: Complex filters for database queries.	19
Problem description:.....	19
Justification of chosen solution.....	19

Decision 6: Showing incomes from slot machines.	19
Problem description:.....	19
Justification of chosen solution.....	20
Decision 7: Values from selectors that depend from others.....	20
Problem description:.....	20
Decision 8: Showing client gains from certain weeks.....	21
Problem description:.....	21

PHOTOGRAPHY'S INDEX

Photography 1 - Domain diagrams	6
Photography 2 - Method createEvent (used in Proxy pattern)	8
Photography 3 - Method processUpdategameForm (used in Proxy pattern)	9
Photography 4 - Method eventsByDay (used in Proxy pattern)	9
Photography 5 - File jsp	10
Photography 6 - Casinotable Entity	11
Photography 7 - Method initUpdateCasTbForm	11
Photography 8 - Casinotables' view	11
Photography 9 - Entities	12
Photography 10 - CasinotableService is calling to CasinotableRepository	13
Photography 11 - Services	13
Photography 12 - Querys in CasinotableRepository.....	13
Photography 13 - Controllers, Validators and Formatters	14
Photography 14 - Jsp files	14

Introduction

Tuvi's Casino is a gambling games center which has among their facilities services such as a restaurant, where visitors can have whatever meal they desire in the moment, some show scenarios, where artists perform their shows on a daily basis, and a lot of gaming tables and slot machines

Due to all this, the management of this casino without a proper website is a hard task in day-to-day.

The goal of our Project is to develop a website with which the admins of Tuvi's Casino will be able to successfully manage their business.

This website is going to contain features such login for each different type of user (admin, worker, and client). Depending on the kind of user that is logged on the website, it will able him to manage different things.

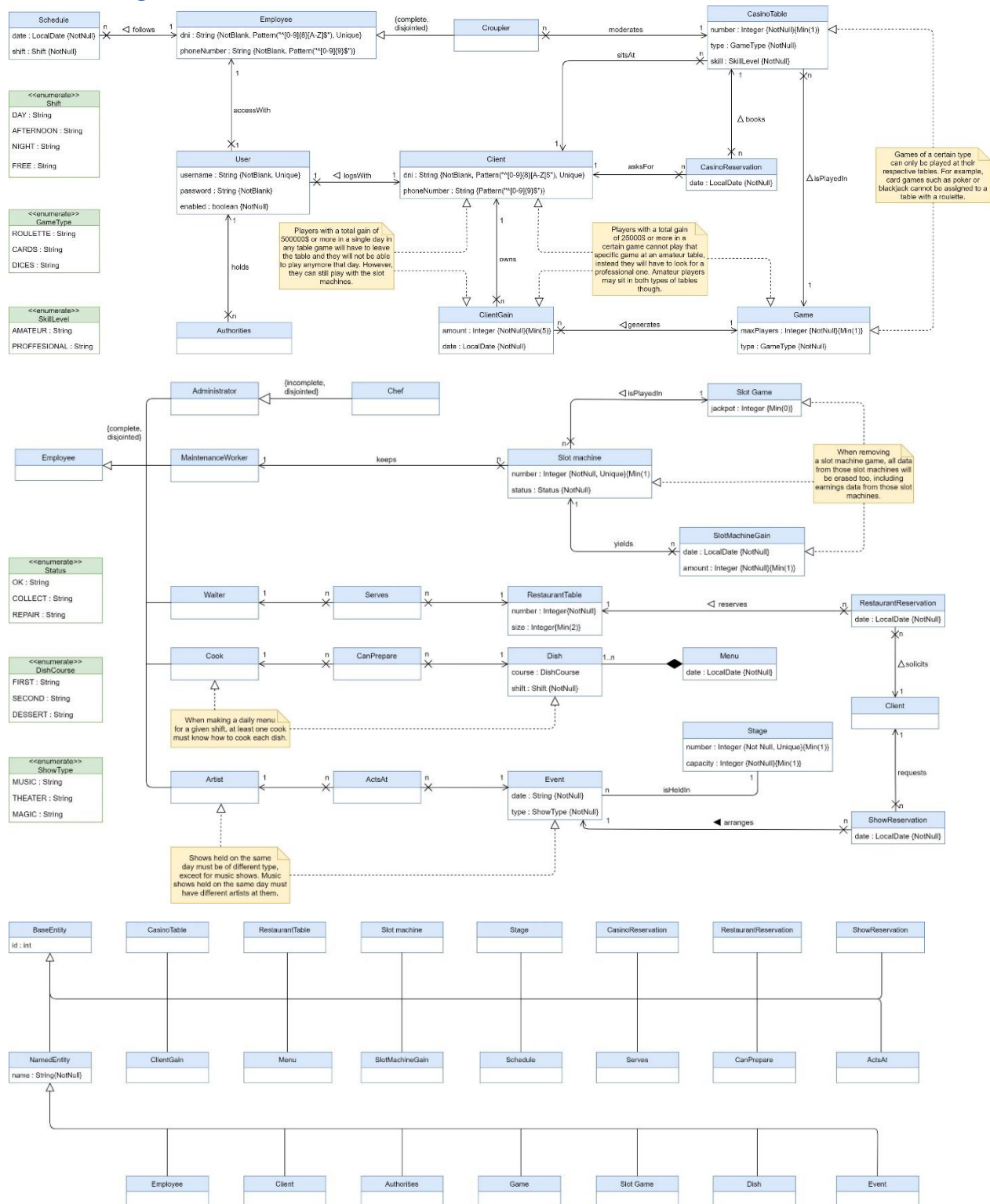
If the user who is logged is a client, the website will show information about his income record, the games he can play and the shows that are going to take place in the casino. Also, the client will be able to reserve table in the Casino's restaurant using this website.

If the user who is logged is an employee, the website will able him to check their shift and where is he going to work, depending on the employee type (this types are described in detail in this document).

In closing, if the user who is logged on the website is an admin, the website will show all the information about the casino, and the admin will be able to manage features such as the employees' shifts or the events distribution and the addition of tables or slot machines.

UML Diagrams:

Domain Diagram



Photography 1 - Domain diagrams

DP1 2020/21
<Nombre Proyecto>

Grupo: GX-XX

Layers Diagram

<https://github.com/gii-is-DP1/dp1-2020-g1-07/blob/master/documents/3rd%20Sprint/DiagramaDeCapas.png>

Design and architectonic patterns applied

Pattern: Proxy (Front Controller)

Type: Design

Application context

We use the proxy design pattern (called Front Controller in Spring). What we do with this is generate a component (the proxy) that takes care of "intermediate" tasks without directly communicating two components. In our case would be the Dispatcher Servlets in charge of fulfilling this function.

The Dispatcher Servlet, complying with the MVC architectural pattern, takes the requests from the page (view) and is responsible for acting as an intermediary between the components of MVC. To do this, first, upon receiving the request, proceeds to make the Handler Mapping. In the mapping, we use annotations of the form:

```
@GetMapping(path="/new")
public String createEvent(ModelMap modelMap) {
    String view="events/addEvent";
    modelMap.addAttribute("event", new Event());
    return view;
}
```

Photography 2 - Method createEvent (used in Proxy pattern)

“@GetMapping”, to specify which route the system should follow. After mapping, the servlet contacts the driver. The driver takes the HTTP/HTML information and extracts it from String's java objects. To do this, Spring provides us with the BindingResult tool.


```
@PostMapping(value =("/{eventId}/edit")
public String processUpdategameForm(@Valid Event event, BindingResult result,
    @PathVariable("eventId") int eventId, ModelMap model) {
    event.setId(eventId);
    if (result.hasErrors()) {
        model.put("event", event);
        return "events/updateEvent";
    }
    else {
        if(eventValidator.eventWithTheSameName_Update(event.getName(), eventId)){
            result.rejectValue("name", "name.duplicate", "El nombre esta repetido");
            model.addAttribute("event", event);
            return "events/updateEvent";
        }
        this.eventService.save(event);
        return "redirect:/events";
    }
}
```

Photography 3 - Method processUpdategameForm (used in Proxy pattern)

After this, the Servlet receives from the controller: the view to be used and the model (Model Map) to be used in it.

```
@GetMapping(path="/byDay")
public String eventsByDay(ModelMap modelMap) {
    String vista= "events/eventsByDay";
    Collection<LocalDate> list=eventService.findAllDates();
    Iterable<LocalDate> dates = list;
    modelMap.addAttribute("dates", dates);
    return vista;
}
```

Photography 4 - Method eventsByDay (used in Proxy pattern)

Once redirected, before reaching the view, a View Resolver is processed, whose task is to specify what type of view we are going to reach. In our case, we work with JSP's. Finally, the Servlet (our proxy) is responsible for sending the ModelMap to the view and printe it. When using JSP, we use the template view, practically HTML, with JSP elements (representation of model attributes).

```
33 <petclinic:layout pageName="menusByDay">
34   <jsp:body>
35     <h2>Menus</h2>
36
37     <div class="control-group">
38       Dates <select id="comboboxDates" name="date">
39         <option selected>Selecciona fecha</option>
40         <c:forEach var="date" items="${dates}">
41
42           <option value="${date}">${date}</option>
43         </c:forEach>
44       </select>
45     </div>
46
47     <div id="tableMenus"></div>
48   </jsp:body>
49
50 </petclinic:layout>
```

Photography 5 - File.jsp

Pattern: Model View Controller (MVC)

Type: Architectonic

Application context

Another pattern that we apply in the application is MVC (due to the characteristic of the framework that we use, Spring) implementing packages that correspond to model, views and controllers. In the application, the model corresponds to entities created that provide information on their attributes, in addition to methods that allow obtaining data when showing them to the user through views or configuring them.

```
@Getter
@Setter
@Entity
@Table(name = "casinotable")
public class Casinotable extends NamedEntity{

    @NotEmpty
    private String name;
    @ManyToOne
    @JoinColumn(name = "game_id")
    private Game game;
    @ManyToOne
    @JoinColumn(name = "gametype_id")
    private GameType gametype;
    @ManyToOne
    @JoinColumn(name = "skill_id")
    private Skill skill;
    @NotNull
    @DateTimeFormat(pattern= "yyyy/MM/dd")
    private LocalDate date;
    @NotEmpty
    @Pattern(regexp="^(?:([01]?\\d|2[0-3]):)?(?:[0-5]?\\d):?(?:[0-5]?\\d)$")
    private String startTime;
    @NotEmpty
    @Pattern(regexp="^(?:([01]?\\d|2[0-3]):)?(?:[0-5]?\\d):?(?:[0-5]?\\d)$")
    private String endTime;
```

Photography 6 - Casinotable Entity

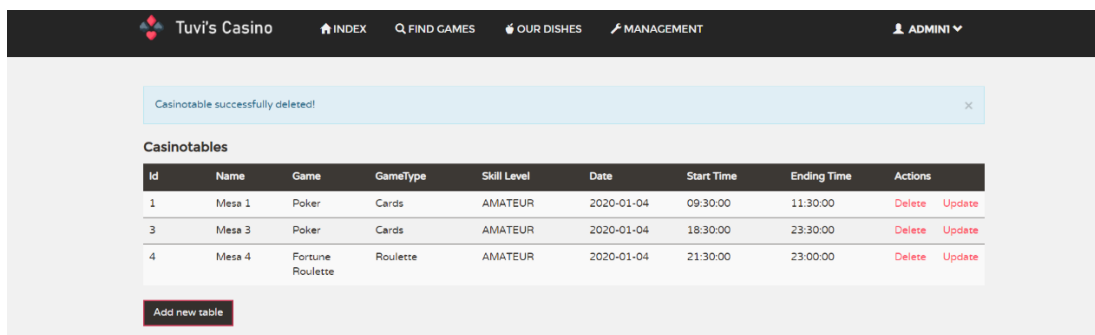
Controllers are the part that allow the application to enter different views or other controllers, in addition to checking the data in validators and configuring data that is displayed.

```
@GetMapping(value = "{casinotableId}/edit")
public String initUpdateCasTbForm(@PathVariable("casinotableId") int casinotableId, ModelMap model) {
    Casinotable casinotable = castableService.findCasinotableById(casinotableId).get();

    model.put("casinotable", casinotable);
    return "casinotables/updateCasinotable";
}
```

Photography 7 - Method initUpdateCasTbForm

Once a controller leads to another view, this is displayed to the user, who can access to the different services of the application.



The screenshot shows the 'Tuvi's Casino' web application. At the top is a navigation bar with links for INDEX, FIND GAMES, OUR DISHES, and MANAGEMENT, along with an ADMIN dropdown. A light blue success message states 'Casinotable successfully deleted!'. Below this is a table titled 'Casinotables' with columns for Id, Name, Game, GameType, Skill Level, Date, Start Time, Ending Time, and Actions. The table contains three rows of data. At the bottom left, there is a red button labeled 'Add new table'.








































Id	Name	Game	GameType	Skill Level	Date	Start Time	Ending Time	Actions
1	Mesa 1	Poker	Cards	AMATEUR	2020-01-04	09:30:00	11:30:00	Delete Update
3	Mesa 3	Poker	Cards	AMATEUR	2020-01-04	18:30:00	23:30:00	Delete Update
4	Mesa 4	Fortune Roulette	Roulette	AMATEUR	2020-01-04	21:30:00	23:00:00	Delete Update

Photography 8 - Casinotables' view

Clases o paquetes creados

PACKAGES | MODEL CLASSES

As previously stated, entities represent the application model, being these classes in the package *Model* inside the application.

- | | |
|--|--|
| >  Administrator.java | >  Schedule.java |
| >  Artist.java | >  Shift.java |
| >  Authorities.java | >  ShowType.java |
| >  BaseEntity.java | >  Skill.java |
| >  Casinotable.java | >  SlotGain.java |
| >  Chef.java | >  Slotgame.java |
| >  Client.java | >  SlotMachine.java |
| >  ClientGain.java | >  Specialty.java |
| >  Cook.java | >  Stage.java |
| >  Croupier.java | >  Status.java |
| >  Dish.java | >  User.java |
| >  DishCourse.java | >  Vet.java |
| >  Employee.java | >  Vets.java |
| >  Event.java | >  Visit.java |
| >  Game.java | >  Waiter.java |
| >  GameType.java | |
| >  MaintenanceWorker.java | |
| >  Menu.java | |
| >  NamedEntity.java | |
| >  Owner.java | |
| >  package-info.java | |
| >  Person.java | |
| >  Pet.java | |
| >  PetType.java | |

Photography 9 - Entities

PACKAGES | CONTROLLER CLASSES

In the previous section we clarified that controllers use validators to verify the proper use of data that is configured in views, but other classes that allow to acquire information found in the database are also used.

In the first place, we will focus on *Service* and *Repository* packages, being these classes of the first package those that call the classes of the second.

```
@Autowired
private CasinotableRepository castabRepo;
@Autowired
public CasinotableService(CasinotableRepository castabRepo) {
    this.castabRepo = castabRepo;
}

@Transactional
public int casinoTableCount() {
    return (int)castabRepo.count();
}
```

Photography 10 - CasinotableService is calling to CasinotableRepository

```
> AdministratorService.java
> ArtistService.java
> AuthoritiesService.java
> CasinotableService.java
> ChefService.java
> ClientGainService.java
> ClientService.java
> CookService.java
> CroupierService.java
> DishService.java
> EmployeeService.java
> EventService.java
> GameService.java
> MaintenanceWorkerService.java
> MenuService.java
> OwnerService.java
> PetService.java
> ScheduleService.java
> SlotGainService.java
> SlotGameService.java
> SlotMachineService.java
> StageService.java
> UserService.java
> VetService.java
> WaiterService.java
```

Photography 11 - Services

```
public interface CasinotableRepository extends CrudRepository<Casinotable, Integer>{

    @Query("SELECT skill FROM Skill skill ORDER BY skill.id")
    List<Skill> findSkills() throws DataAccessException;

    @Query("SELECT gtype FROM GameType gtype ORDER BY gtype.id")
    List<GameType> findGameTypes() throws DataAccessException;

    @Query("SELECT game FROM Game game ORDER BY game.id ")
    List<Game> findGames()throws DataAccessException;

    @Query("SELECT game FROM Game game WHERE game.gametype.id = :id ORDER BY game.id")
    public List<Game> findGamesByGameType(@Param("id") int id);

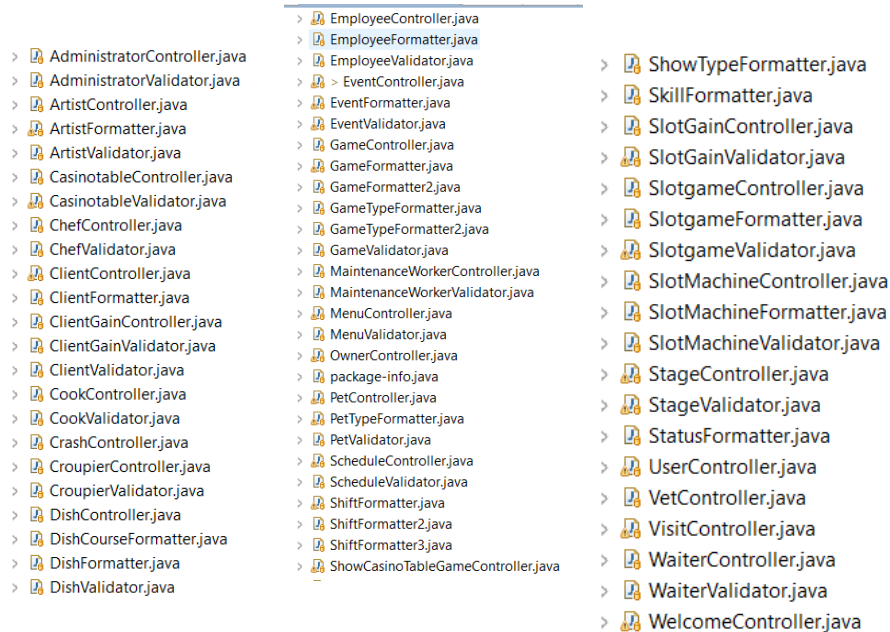
    @Query("SELECT casinotable FROM Casinotable casinotable ORDER BY casinotable.id")
    public List<Casinotable> findCasinoTables();

    @Query("SELECT DISTINCT date FROM Casinotable")
    public List<LocalDate> findAllDates();

    @Query("SELECT casinotable FROM Casinotable casinotable where casinotable.date = :date ORDER BY casinotable.id")
    public List<Casinotable> findCasinoTablesByDate(@Param("date") LocalDate date);
}
```

Photography 12 - Queries in CasinotableRepository

Classes of *Service* package are used in controllers implemented in the *Web* package, where they also find validators that allow the verification of data.



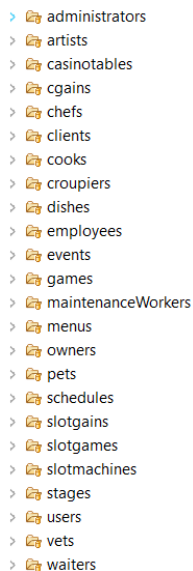
- > AdministratorController.java
- > AdministratorValidator.java
- > ArtistController.java
- > ArtistFormatter.java
- > ArtistValidator.java
- > CasinotableController.java
- > CasinotableValidator.java
- > ChefController.java
- > ChefValidator.java
- > ClientController.java
- > ClientFormatter.java
- > ClientGainController.java
- > ClientGainValidator.java
- > ClientValidator.java
- > CookController.java
- > CookValidator.java
- > CrashController.java
- > CroupierController.java
- > CroupierValidator.java
- > DishController.java
- > DishCourseFormatter.java
- > DishFormatter.java
- > DishValidator.java
- > EmployeeController.java
- > EmployeeFormatter.java
- > EmployeeValidator.java
- > EventController.java
- > EventFormatter.java
- > EventValidator.java
- > GameController.java
- > GameFormatter.java
- > GameFormatter2.java
- > GameTypeFormatter.java
- > GameTypeFormatter2.java
- > GameValidator.java
- > MaintenanceWorkerController.java
- > MaintenanceWorkerValidator.java
- > MenuController.java
- > MenuValidator.java
- > OwnerController.java
- > package-info.java
- > PetController.java
- > PetTypeFormatter.java
- > PetValidator.java
- > ScheduleController.java
- > ScheduleValidator.java
- > ShiftFormatter.java
- > ShiftFormatter2.java
- > ShiftFormatter3.java
- > ShowCasinoTableGameController.java
- > ShowTypeFormatter.java
- > SkillFormatter.java
- > SlotGainController.java
- > SlotGainValidator.java
- > SlotgameController.java
- > SlotgameFormatter.java
- > SlotgameValidator.java
- > SlotMachineController.java
- > SlotMachineFormatter.java
- > SlotMachineValidator.java
- > StageController.java
- > StageValidator.java
- > StatusFormatter.java
- > UserController.java
- > VetController.java
- > VisitController.java
- > WaiterController.java
- > WaiterValidator.java
- > WelcomeController.java

Paquete Web, que contiene los controladores y los validadores

Photography 13 - Controllers, Validators and Formatters

PACKAGES | VIEWS CLASSES

For views we use different .jsp files that allow the user to see the content they prefer, as it was previously shown with the *"/casinotables"* view



- > administrators
- > artists
- > casinotables
- > cgains
- > chefs
- > clients
- > cooks
- > croupiers
- > dishes
- > employees
- > events
- > games
- > maintenanceWorkers
- > menus
- > owners
- > pets
- > schedules
- > slotgains
- > slotgames
- > slotmachines
- > stages
- > users
- > vets
- > waiters

Photography 14 - Jsp files

Design decisions

Decision 1: Using javascript to get the element id to complete the edit url

Problem description:

We could not get the element id of the element that was described on the form

Solution 1.a: Examining the controller to get the element id

Advantages:

- It is contained on the controller

Disadvantages:

- Harder to code and takes more time

Solution 2.b: Using javascript to get the element id.

Advantages:

- This part of the code contained in the jsp
- Easier to code and understand

Disadvantages:

- It is not included at the backend

Justification of chosen solution

We consider that understanding and simplifying the code when it is possible is very important. That is because we have chosen the solution 2.b

Decision 2: Tests in controllers and services

Problem description:

The aim when doing tests is to check the correct functioning of the different components from the project without needing to execute the application or doing tests over the running application.

Solution 2.a: Make tests by doing @Autowire of services and controllers.

Advantages:

- No need for mock-ups.
- Code is simpler.
- The data is already prepared (they are the same that are used when the app runs).

Disadvantages:

- Is no different to doing a test directly on the app.

- Is possible to forget doing some tests due to working with already existing data.
- If there are components that are used by controllers/services that should not be using them, we would not be able to notice the error.

Solution 2.b: Make the service tests by mocking the needed repository and make the controller tests by mocking the needed service. Validators should not be mocked.

Advantages:

- Total independence between tests and actual application.
- All necessary tests for the app's correct functioning are carried away, if any of them is missing, the other existing tests would result in failure.
- Allows to make tests over custom data (simpler data if desired).
- Allows to test the modularity between project components.

Disadvantages:

- More code.
- The data must be prepared prior to the tests

Justification of chosen solution

We have chosen the solution 2.b given that it was very important for the project a total independency between the tests and the application. Also some errors were discovered when using some specific components (mostly formatters) that would not have been noticed if tests would not check the modularity between different components.

Decision 3: Managing the heritage relation from "Employee" entity.

Problem description:

The system must hold 7 types of employee, even though they have similar properties between them.

Solution 3.a: Make a class "Employee" with an enumerate attribute to determine which type of employee it's referring to.

Advantages:

- Only one entity is created.
- No heritage relations between entities which may lead to problems.

Disadvantages:

- It's not a correct model of the real situation. If the client wants to specify whether one employee is a waiter or a croupier, then the system should have different entities for each one.
- A lot of restrictions would have to be made in order to maintain relationships between "Employee" and other entities. For example, only croupiers would attend casino tables, so when an employee is assigned to a casino table there should be a restriction that forbids other kinds of employee, like chefs, to be related to a casino table.

- All kinds of employee are the same class, so it's not possible to add more information or functionality to only one of them easily.
- Even though there's only one entity, we still need several controller and service methods for each kind of employee. This makes those classes remarkably longer, which makes it hard to debug and test.

Solution 3.b: Make a class for each type of employee, and a father class "Employee".

Advantages:

- All classes share attributes, but are independent entities, so future changes to these classes are possible.
- Each entity has its own controller and service, which makes it simple to test and accomplishes both coupling and cohesion principles.
- Much more simple relations between employees and other classes.

Disadvantages:

- All entities have almost identical classes (service, controller, validator, etc), so a lot of code could be reused.
- Deleting or modifying the id of the father class without modifying the child class leads to inconsistencies.

Justification of chosen solution

We decided the 3.b option in the first place because it was a better modelling choice and because we weren't completely sure if the different employees could have more data, so it seemed safer than making one big class and then having to split it because of a small change.

Decision 4: Validation subsequent to @Valid in

Problem description:

When doing an object validation, this object does not have an id yet, so when wanting to check additional restrictions (specially those related with checking if the same elements already exist in the database), it was not possible to do so due to that id not existing. Making an additional validation needed.

Solution 4.a: Make the additional validation only when creating the object and not in modification, where the id is needed.

Advantages:

- Only valid function would be necessary.

Disadvantages:

- In case the rule is broken, an error would appear in the database which would cause the app to stop the process.
- The form validation would be incomplete.
- Different validation when creating and updating.

Solution 4.b: Have an automatic and not modifiable field corresponding to the object's id.

Advantages:

- Does the additional validation in the valid function.

Disadvantages:

- Requires that more elements are added to the form.
- In creation, the id field would be an empty field or an id should be created prior to the form (without letting spring manage it).
- Spring must use the id that is given by us, and not the one it creates.

Solution 4.c: Use the setId that is already on the updates and make the validation afterwards, when the object has an id, by making a call to the function in the validator.

Advantages:

- No elements are added to the form.
- No modification of valid function.
- Error is captured and a user-friendly message is shown.
- It allows more Independence between validations.

Disadvantages:

- More code.
- Updates have more conditions in them.

Justification of chosen solution

We have chosen the solution 4.c because it was easy to implement and did not interfere with the valid function. Also, the additional validations are done the same way in creation and updates, even when it could be done only with the valid function while creating an object. This was done so post functions of creation and update are more alike and to keep the valid function the same for both of them.

Decision 5: Complex filters for database queries.

Problem description:

Some repository classes may need queries that can't be written as a method name notation.

Solution 5.a: Making a normal query in the repository class, then applying filters in the service class.

Advantages:

- Much easier to do with Java 8 streams.

Disadvantages:

- Less efficient.
- More code.
- "Dirty" solution.

Solution 5.b: Using JPA custom queries.

Advantages:

- More efficient and clean.
- Intended way of solving this problem.
- Spring automatically checks custom queries, so there's no need of unit tests.

Disadvantages:

- JPQL syntax must be learned.

Justification of chosen solution

At first instance, we just went with solution 5.a for really complex queries because it was closer to our Java knowledge and we really didn't know otherwise. With time, we instead used 5.b, as it was a more appropriate solution.

Decision 6: Showing incomes from slot machines.

Problem description:

Each slot machine has a different set of gains with each day they are in use. In other words, each day a slot machine earns a different amount of money that is saved by the system. The idea is to be able to show the gain of one slot machine from one specific day.

Solution 6.a: Make an additional view for each slot machine where all its gains are listed, or with a selector to pick the date.

Advantages:

- Easy to code.
- Is similar to other code that is already in the project.

Disadvantages:

- One must change views to view the gains of a slot.
- Is not original, just another list of numbers.
- Uncomfortable if someone wants to check different amounts from different slots

Solution 6.b: Add a selector with dates to each slot machine in the slotMachineList view, so when selecting the date of a gain of an specific slot, the amount that it earned that day is shown in the next column.

Advantages:

- Easy to use.
- Comfortable for the user.
- Does not show unimportant information to the user.
- All the information is in one view.

Disadvantages:

- Very complex ajax coding (a selector for each slot with different options in each selector).
- Needs more functions in the controller/service to work.

Justification of chosen solution

We have chosen the solution 6.b, although we knew it was quite a difficult task to perform, we faced it as a fun challenge and after quite a bit of research and not few mistakes we managed to get the most optimal solution for this problem working. Being the coding difficulty the only disadvantage, after we solved it, all that was left were advantages for the final user.

Decision 7: Values from selectors that depend from others.

Problem description:

Depending on the shift we are consulting, there will be certain dishes available. The website, initially, was not able to show the selected dishes for that shift. It just was showing the entire list of dishes. We can also find this kind of dependencies in other classes.

Solution 7.a: Dividing the section into small views depending on the shift we are selecting.

Advantages:

- Easier to code. We don't have to change anything related to the selectors.

Disadvantages:

- We have to create more views.
- It's not comfortable for the user
- Takes more time

Solution 7.b: When we select the shift, using AJAX will make the selector that contains the list of available dishes change.

Advantages:

- Easier for the user. The user will be able to check all shifts in one view.
- AJAX is specially made for these cases.

Disadvantages:

- Harder to code than the previous one.

Justification of chosen solution

We have chosen this solution because we think that the solution that is best for the user is the one that may be chosen. We have already used AJAX to solve previous problems, we found it natural to apply solution 7. b.

Decision 8: Showing client gains from certain weeks.

Problem description:

The system should be able to show clients their gains for a certain week they select.

Solution 8.a: Reloading the page each time a week is selected.

Advantages:

- Easy to code, it's just a form that redirects to itself.

Disadvantages:

- Reloading the page so many times makes it slow to navigate.

Solution 8.b: Using AJAX for loading data each time a week is selected.

Advantages:

- Faster.
- AJAX is specially made for these cases.

Disadvantages:

- Harder to code than the previous one, we need a controller method and a Javascript function.

Justification of chosen solution

As we already used AJAX to solve previous problems, we found it natural to apply solution 8.b. Even though it was harder, we had the experience and it's a better solution.