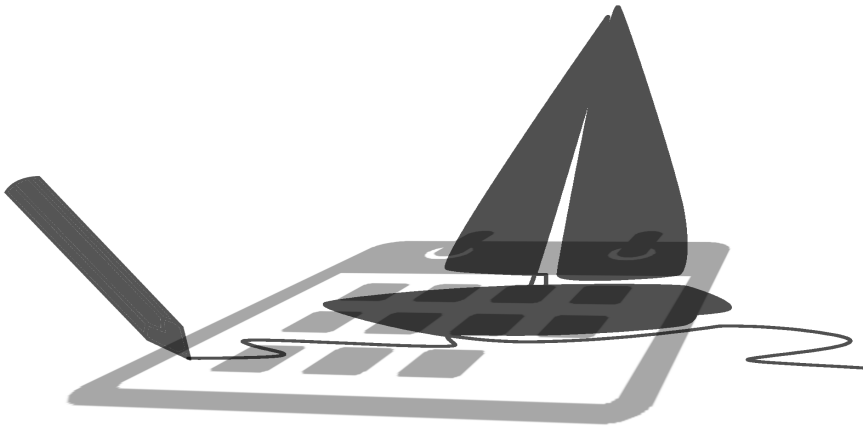
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Electrónica e Telecomunicações e de Computadores



Planeador de Viagem em Veleiro - Logística

Gonçalo Ferreira de Azevedo

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Professor Doutor Paulo Manuel Trigo Cândido Da Silva
Engenheiro Hugo Trovão

Júri:

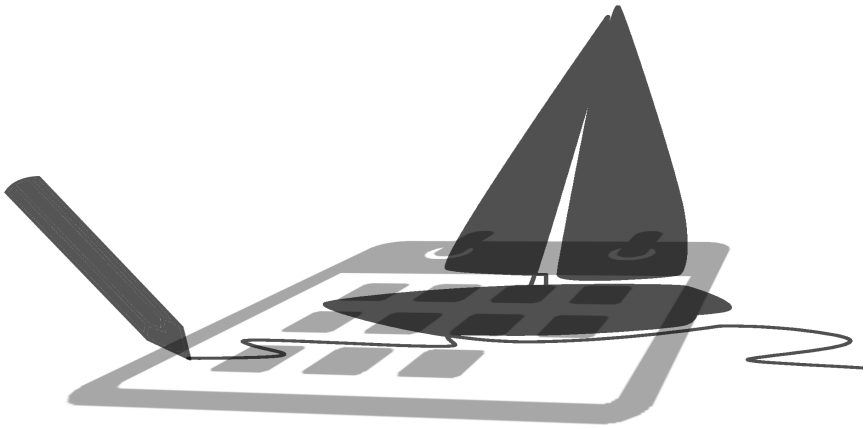
Presidente: Professor Doutor Carlos Jorge De Sousa Gonçalves

Vogais: Professor Doutor Rui Manuel Feliciano De Jesus
Professor Doutor Paulo Manuel Trigo Cândido Da Silva

October, 2022

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Electrónica e Telecomunicações e de Computadores



Planeador de Viagem em Veleiro - Logística

Gonçalo Ferreira de Azevedo

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Professor Doutor Paulo Manuel Trigo Cândido Da Silva
Engenheiro Hugo Trovão

Júri:

Presidente: Professor Doutor Carlos Jorge De Sousa Gonçalves

Vogais: Professor Doutor Rui Manuel Feliciano De Jesus
Professor Doutor Paulo Manuel Trigo Cândido Da Silva

October, 2022

À minha família e amigos

Acknowledgments

I would like to express my thanks towards both of my thesis advisors, Paulo Trigo and Hugo Trovão, for assisting me through this project and for the guidance they provided, everything from ideas, questions, solutions, and any other feedback.

I would like to thank both my colleagues and those I've worked with, for helping me build up the knowledge and experience that has gotten me this far.

Lastly, I would like to thank my family for the support that they've showed me, I owe them everything for making me the person I am today.

Abstract

This work focuses on the logistics' subject of planning a sailing trip. The logistics' subject is one of two parts that make up the trip planning, the other part being the nautical/navigation subject, which deals with topics such as tides, weather, and the route specification.

The logistics of a sailing trip focuses on the specification and distribution of tasks to be executed throughout the trip, information about the nourishment (daily food menu), including ingredients and their weight, products and their respective costs. Although one can find several tools capable of designing routes, calculating fuel, looking up weather, sea currents, there aren't many tools that help in the crew-shift optimization considering user-defined constraints (e.g., mandatory resting and cooking periods), nourishment considering ingredient costs (e.g., availability, cost and weight) and other logistics.

The project was made with the goal of providing users with the necessary tools to efficiently build a travel plan, from task creation, to automatic planning of task schedules, menus and product lists, checklists, among others. An additional goal is the ability to reuse existing plans built by other users.

The tools are built through constraint optimization (using open-source libraries), and made available through a web application and a mobile application. All in order to allow users not only quick access to their plans, but a greater control over them, with the ability to collaborate with each other to build plans, or even search and reuse previous plans.

Keywords: sailing, planning, django, constraint programming, optimization, web application, mobile application

Resumo

Este trabalho foi realizado no âmbito da Tese final de Mestrado em Engenharia Informática e de Computadores, e incidiu sobre o assunto de logística do planeamento de uma viagem de veleiro. A lógistica é uma de duas partes que constituem o planeamento da viagem, send a outra parte a secção de náutica ou navegação, que aborda temas como as marés, a meteorologia, o trajeto náutico.

A logística de uma viagem à vela, foca-se na especificação e distribuição de tarefas a realizar, informações sobre a alimentação a concretizar, incluindo ingredientes e os respetivo peso, os produtos e custos associados. Apesar de se encontrar várias ferramentas capazes que possibilitam o desenho de trajectos, calculo de combustível, meteorologia, correntes marítimas, não existem muitas ferramentas que auxiliam na optimização de tripulantes por turno, através de restrições definidas pelo utilizador (exemplo, descanso obrigatorio e períodos onde se deva cozinhar), formulação de um plano de alimentação considerando informações dos ingredientes (peso, preço e disponibilidade) e outra logística.

O projecto foi realizado com a idea de fornecer aos utilizadores as ferramentas necessarias para de uma forma eficiente, conseguirem construir um plano de viagem, desde criação de tarefas, planeamento automatico de horários de tarefas, ementas e listas produtos, checklists, entre outros. Com o objectivo adicional da capacidade de reutilização de planos já existentes construídos por outros utilizadores.

As ferramentas são construidas através da utilização de bibliotecas de optimizações com restrições, e disponibilizadas através de uma aplicação web e uma aplicação mobile, de forma a permitir aos utilizadores não só acesso rápido, mas um maior controlo sobre os planos, com capacidade para colaborarem entre si para construir planos, ou mesmo pesquisar e reutilizar planos posteriores.

Palavras-chave: viagem à vela, planeamento, django, programação com restrições, optimizações, aplicação web, aplicação mobile …

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

This project was developed in the context of the thesis of the Master's degree of the course "Mestrado em Engenharia Informática e de Computadores" (MEIC) from the institution "Instituto Superior de Engenharia de Lisboa (ISEL)"

This project's purpose is to develop tools and functionalities capable of assisting the users and allow them to more efficiently prepare for sailing trips, or to completely skip the trip drafting process in order to utilize someone's previously executed journey plan.

The solution gives users tools that allows them to manage important subjects like tasks, meals, ingredients, products and even other crew members. While also allowing a team of users to cooperate managing a plan, and eventually, if desired, share that plan for future users to re-use.

1.1 Motivation

Sailing is the use of the wind, through the control of sails, to propel a vessel across the surface of the water to a chosen destination.

Historically, sailing has been present in several civilizations, the oldest record of a sail was found in an Egyptian vessel from 3500 B.C.

At one time it was one of the most important forms of travel, trade, and battle, but it fell into disuse as steam navigation began to appear.

Today it is used recreationally.

A sailing trip poses many challenges to the crew at sea, so it is of utmost importance that the planning and execution of the trip be carried out with a certain amount of rigor.

After all, once in the sea, there is no way to solve any issues that haven't been accounted for, unlike on land.

The tools built in this project provide functionalities, so that users can plan their trips in detail, while also allowing for sharing of these plans with other users as a way to provide less tech savvy users with simple to use pre-made plans.

The tools are made available through a simple web application, whose sole purpose is to serve as an interface to access these tools.

1.2 Objectives

The project's focus is on the logistics part of sailing, the challenges identified will relate more to the preparation, organization and planning phases of the trip, as well as assistance for the plan's successful execution.

The tools the project aims to supply are meant to mainly address the following problems and objectives:

- **Planning**
  - **Team-Work** - the solutions should allow entire teams, given the necessary permissions, to edit plans they belong to, which allows for cooperation and feedback between crew members during the plan creation.
  - **Work-Reuse** - created plans should have options to allow for the sharing of plan templates, which allows for less tech knowledgeable users to reuse plans without needing to understand actual task creation or any other complex systems.
- **Meals and menus**
  - **Meals** - it is important to have an idea of what dishes will be made during the trip, not only because of the ingredients, but also as a way to better organize the trip and avoid "in the moment" decisions, as delaying the question of "what to make" to the moment of the trip itself could lead to confusion and lack of preparation.

  - **Ingredient Planning** - knowing which ingredients to bring also allows us to create a plan to keep track of how many ingredient to acquire, their usage and their weight, which are all useful thing to know for a trip.
  - **Translation** - because some menu/ingredient information is not available in multiple languages, the usage of translation tools allows staff to freely select menus from English recipe APIs.

- **Supplies and products**
  - **Products** - by translating the ingredients into Portuguese, it is possible to map out what respective retail products contain the necessary ingredients for the trip and obtain said product information from retail websites.
  - **Pricing** - product information also allows for the creating of a price table and optimize solutions in order to minimize costs.
  - **Weight** - similarly to the price table, a weight table can also be created and optimized for, which allows users to lighten the ingredient load if needed.

- **Tasks and scheduling**
  - **Identifying** - during a sailing trip, there are various tasks that should be completed, from being on lookout, to cooking, and even boat maintenance. Having a good idea what to do during a trip is important as negligence could lead to undesired result, as such, identifying what to do is of extreme importance.
  - **Qualifications** - each individual's capabilities should be taken into account as not every person is qualified to execute certain tasks.
  - **Scheduling** - after identifying the tasks, it is important to distribute them among all crew members, while this might sound simple at first, the distribution must take into account multiple factors, if the crew member is asleep, if the crew member is busy with another task.
  - **Execution** - there should also be a way to keep track of which tasks to undertake, and when to do them, as well as a way to tell which task has been completed. Keeping track of tasks is done to avoid forgetting or accidentally not completing certain tasks.

As a disclaimer, recall that the project does not focus on problems and solutions related to other subjects related with sailing trip planning, such as:

- **Itinerary**
  - **Route** - the nautical route to undertake during the trip.
  - **Duration** - expected duration obtained from the route information combined with vessel capabilities.
- **Conditions**
  - **Weather** - information regarding weather conditions and time frames.
  - **Sea Tides** - information directly related to the sea itself.
  - **Winds** - it's important as sometimes, fuel will be needed for the trip when the wind does not favor the journey.

1.3 Contributions

In an attempt to create a platform to fill the hole left by the lack of tools useful to help plan out the logistics of a sail trip, this project has resulted in both a web server and a mobile application.

The web server is responsible for distributing the web content as well as handling requests made by the mobile application. The web server also houses the code used to generate schedules and any other tools used in this project.

The schedule generation was coded to be used in a generic way, this in order to be able to adapt to a multitude of situations, as such, if need be, it could be used to create schedules for other activities unrelated to sailing. The same can also be said for the meal planner. Although the reason behind this is to enable the solution to adapt to different circumstances, the end result was the creation of two tools that have a wide possible range of various usages.

Both the code for web server (and all it encompasses) as well as the code for the mobile application can be found in https://github.com/a43554/sail_web_server and https://github.com/a43554/sail_mobile_client, respectively.

An article related to the project was also drafted. The article summarises the information that can be found in this document, and thus can also be seen as a less extensive version of this document, while also containing all key information.

1.4 Document Structure

This document is organized in five chapters:

- **Introduction** - this chapter, which briefly describes the problem and the solution.
- **Related Work** - chapter dedicated to other solutions for sailing and related problems.
- **Technology and Fundamentals** - the wide range of possible technologies that are relevant to the current project.
- **Proposed Solution** - the chosen approach to the problem and the details that define the process and structure of the solution.
- **Conclusion** - the final thoughts on the project, both retrospective and future outlook.

2

Related Work

The related work can be split into two parts.

On one hand, the solutions that aim to directly to solve broad problems in the context of sailing. On the other hand, the solutions that are not specifically related to the topic, but are instead related to an approach to solve one of the specific problems raised in planning a sailing voyage, such as, solving scheduling problems.

2.1 Broad Sailing Solutions

This subsection presents solutions pertaining directly to the subject of planning a sailing trip and executing it.

2.1.1 Package/Templates (Tourism)

The idea behind a package or template approach is to present clients with templates of already thought-out trips, with possible detail customization. We can observe this model in travel and/or tourism agencies.

Travel agencies aim to sell customers the trip itself, and the degree of freedom that customers are allowed to exercise over the customization of routes, itineraries, duration, and vessels may vary. For example, "SailingEurope" [27] provides the user with the ability to choose or design their own route, select the vessel and the participants,

another example, "Sunsail" [31] offers several different sailing travel packages. For example, they offer vacations with a set of sailing activities for family, romantic, group, and others.

This approach allows for most of the work needed for trip preparation to be moved from the client's side to company's, the less customization, the simpler it is to complete the trip preparation.

The biggest advantage to be seen in the package approach is the ability to easily create and reuse trip plans according to users' wishes.

2.1.2 Sailing Route Planning

There are plenty of route planning tools for sailing on the web, but despite being powerful tools, they focus more on the navigation, route and weather parts of sailing trips, which are not the subject of this project.

Even so, they are still solutions capable of indicating beneficial data on the logistics side, for example, some tools like "FastSeas" [13] or "Windy.app" [32] indicate the number of days it would take to travel the planned route, which is very helpful data.

2.1.3 Sailing Guides

Book guides or other online guides are useful tools because they provide crew members with a wealth of information related to sailing.

From information about the tasks and activities to be done on a sailing trip, to checklists to keep track of what needs to be done during the trip and how to prepare for it.

Books like "The Handbook of Sailing" [1], "The Complete Sailing Manual" [2] or "The Complete Sailor" [3] provide information related not only to sailing, but also about equipment maintenance, procedures, day to day activities while on board, regulations, among other important information.

2.2 Approaches to Specific Problems

This subsection will present solutions to problems not related to the topic itself, but solutions related to problems that can be encountered in sailing voyages.

Task Planning

A scheduling problem is described as:

"More generally, scheduling problems involve jobs that must scheduled on machines subject to certain constraints to optimize some objective function."[28] The authors describe quite a few approaches to these problems and their respective algorithms, some of the approaches consist notably of:

- **Priority Rules** - assign each *task* a priority value according to an optimization criteria. Assign the *tasks* to the *machines* by order of priority.
- **Sophisticated Greedy Approaches** - these types of approaches go beyond just ordering the tasks by priority, it takes into account other *tasks* in a way that it results in a solution being gradually build upon from nothing.
- **Matching and Linear Programming** - it consists of taking central themes of combinatorial optimization and using them to solve planning problems.

It's easy to see where planning problems fit into the context of sailing as it is necessary for *people (machines)* on board to complete day to day *tasks* following certain *rules (constraints)*.

3

Technologies and Fundamentals

This chapter will cover in detail the challenges of meeting the objectives presented in the logistical section of planning a sailing voyage and some of the respective tools and strategies employed for solving these challenges.

3.1 Constraint Satisfaction and Optimization

A constraint satisfaction problem is characterized as a mathematical question where there is a set of objects, in this case variables, and their respective states, which, together have to fulfill a set number of conditions or restrictions.

Each variable will take a value from a set of values in a known domain. The constraints are conditions that the variables must meet in order to solve the problem.

The implementation consists in writing a problem with variables and constraints and construct a model, so that it generates all possible solutions that satisfy all the conditions.

An optimization problem consists in applying an optimization function, which will assign a value to the set of states of a solution, in order to quantify how optimal a solution is, thus allowing the solutions to be categorized and the best possible solution to be extracted from the set of all valid solutions.

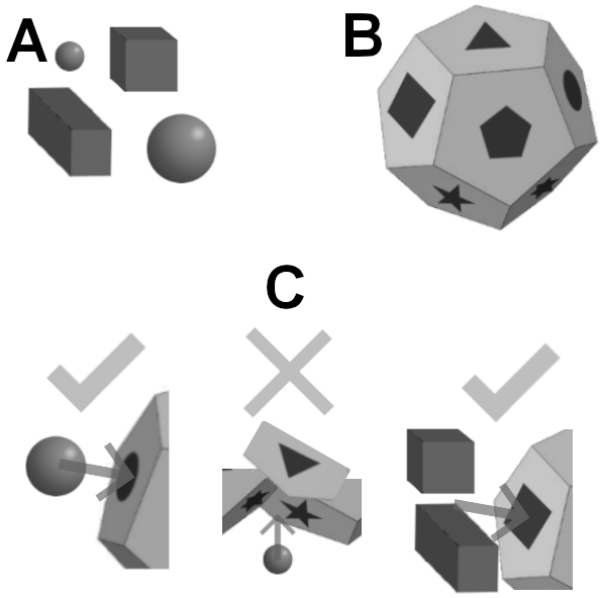


Figure 3.1: Example of a constraint problem.

In the above example, solids (A) must be inserted through figure shaped holes in the toy (B), adhering to some logic (C). In this case, the solids are the variables, the hole they will be inserted in are the states or values that the variables will take, and they must adhere to logic, the larger solids can only fit in their respective holes, those are restrictions. The smaller sphere however, can go into any hole it fits in, as such, the respective restriction is that the small sphere cannot be inserted into either star hole nor the triangle hole because those openings are too small.

3.1.1 Context

The use of constraint satisfaction techniques is a complex and interesting subject of study. It is currently applied with success to many domains such as scheduling, planning, vehicle routing, configuration, networks, and bioinformatics. [4]

One of the constraint satisfaction problems identified in this project is the conflicts created by incompatibilities between task assignments, shift scheduling and meal times.

If we were to manually assign crew members to tasks, there are a number of immediately visible factors to account for: "is the person awake? If yes, are they qualified to perform this task? If yes, are they busy with another task? Can they perform both during the same shift? If they can't, can we assign someone else? or do we reschedule either task? Won't that create more conflicts?"

As such, in order to deal with such problems, the current project tackles planning and scheduling through the usage of constraint satisfaction techniques.

3.1.2 Tools

In this section two tools will be presented that allows for a problem to be translated into code and also apply the necessary constraints and optimization functions in order to obtain various solutions to the problem.

Google OR-Tools

The Google OR-Tools [23] is an open source tool used for optimization of combinations. It provides models capable of solving constraint satisfaction problems, vehicle routing problems, linear optimization problems, and others.

The library is available in Python [25], C++ [7], .NET [11] and Java [17].

For solving constraint programming problems, this tool works by providing methods that allow for the creation of variables of various types and the respective constraints to apply to the variables.

An example on the library usage can be found below. The example consists of using 2 variables (x and y) which can assume values of either 0 or 1 and are restricted by two constraints, the first one forces the sum of both variables to be 1 or less. The second constraint forces both variables to have the same value.

For this specific example, the number of constraints and their complexity is simple enough that the set of all possible solutions can be written out as such:

$$\{(x,y)|x \in \mathbb{Z}^*, y \in \mathbb{Z}^*, 0 \leqslant x \leqslant 1, 0 \leqslant y \leqslant 1, x = y, x + y \leqslant 1\}$$

It can also be written and solved through an equation system:

$$Domains:$$
$$0 \leqslant x \leqslant 1, with\ x \in \mathbb{Z}^*$$
$$0 \leqslant y \leqslant 1, with\ y \in \mathbb{Z}^*$$

$$Constraints: \left\{ \begin{array}{c} x = y \\ x + y \leqslant 1 \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{c} x = y \\ 2x \leqslant 1 \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{c} x = y \\ x \leqslant \frac{1}{2} \end{array} \right\}$$

$$Solution: \left\{ \begin{array}{c} x = y \\ x \leqslant \frac{1}{2} \end{array} \right\} \Rightarrow \left\{ \begin{array}{c} x = y \\ x < 1 \end{array} \right\} \Rightarrow \left\{ \begin{array}{c} x = y \\ x = 0 \end{array} \right\} \Rightarrow \left\{ \begin{array}{c} y = 0 \\ x = 0 \end{array} \right\}$$

Listing 3.1: Python implementation of a problem using Google OR-Tools

```
# Declare the model.
model = cp_model.CpModel()

# Create the variables with values a domain of values [0, 1].
x = model.NewIntVar(0, 1, 'x')
y = model.NewIntVar(0, 1, 'y')

# Apply the constraint "variables sum must not be larger than one".
model.Add(x + y <= 1)
# Apply the constraint "variable values must be equal".
model.Add(x == y)

# Create and run the solver.
solver = cp_model.CpSolver()
status = solver.Solve(model)

# Obtain the value of variable 'x', in this case result will be 0.
x_value = solver.Value(x)
```

The problem, formulated in Listing 3.2, is an actual example of a simple scheduling problem:

"The user A is assigned 2 out of 4 possible shifts, but no shifts may be assigned sequentially."

In order to represent the problem, we need to identify the 3 necessary components to represent the scheduling problem:

- Variables - which correspond to shifts, each variable represents a single shift. By storing each variable in an array, it is possible to create a list of shifts, where each list index corresponds to a shift number.

- Values - which corresponds to whether or not the shift is taken. Using a boolean value (domain of values [0, 1]), where 0 means the shifts isn't occupied and 1 means that the user is assigned to that shift.
- Constraints - which correspond to the "rules" the solution must obey. There are 2 rules or constraints:
  - 2 out of the 4 shifts must be assigned to user A.
  - The user cannot be assigned to 2 shifts in sequence.

Listing 3.2: Scheduling example

```
# Declare the model.
model = cp_model.CpModel()

# Create a list of variables with boolean values. (Domain [0, 1]).
# Each index of the array represents the shift number.
# And each value of the domain represents if that shift
# is taken (value = 1) or not (value = 0) by user A.
possible_shifts = [
    model.NewBoolVar('task_sh_0'),
    model.NewBoolVar('task_sh_1'),
    model.NewBoolVar('task_sh_2'),
    model.NewBoolVar('task_sh_3')
]

# The length of the array is the total number of shifts (4).
total_possible_shifts = len(possible_shifts)

# Add a rule (constraint):
# The user must undertake at half (2) of the total shifts (4).
model.Add( sum(possible_shifts) == (total_possible_shifts / 2) )

# Add a second rule:
# Shifts must NOT be assigned sequentially.
for idx in range(total_possible_shifts - 1):
    # Force the sum of the current and next shift to be one or less.
    # This makes it impossible for two shifts to be assigned in sequence.
    model.Add( possible_shifts[idx] + possible_shifts[idx + 1] <= 1 )

# Create and run the solver.
solver = cp_model.CpSolver()
status = solver.Solve(model)

# The possible solutions/outputs can be seen below:
# [1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 0, 1]
```

The library also provides other useful functions like OnlyEnforceIf, which can be used to disable or enable constraints based on the value of other variables.

An example where OnlyEnforceIf can be applied can be seen below, where the value of the variable b will change according to the values of variables x and y, in order to simulate the AND logic gate.

Listing 3.3: Example, of one of the specific functions that the library possesses

```
...
# Create three boolean variables (domain of values [0, 1]).
x = model.NewBoolVar('x')
y = model.NewBoolVar('y')
b = model.NewBoolVar('b')

# .OnlyEnforceIf(<var>), is a function that is applied to a
# constraint that makes it so that the constraint is only
# created if the <var> value is true (<var> = 1).

# Constraint #A - Make it so that, if b is true, then the sum of
# both x and y must be equal to 2.
model.Add(x + y == 2).OnlyEnforceIf(b)

# Constraint #B - Make it so that, if b is false, then the sum
# of both x and y must be lower than 2.
model.Add(x + y < 2).OnlyEnforceIf(b.Not())

# Thanks to this usage of the "OnlyEnforceIf" function, we have
# successfully created another variable 'b' that will take the
# value of the result of the AND gate between 'x' and 'y'.

# As we know that if 'b' is true, constraint A will be enforced
# (and B will not be enforced) both values of 'x' and 'y' must
# be one.

# And if 'b' is false, then constraint B will be enforced
# (and A will not be enforced) both values of 'x' and 'y' must
# be either both 0, or only a single variable is 1 and the other is 0.
...
```

PuLP

PuLP [24] is another tool that provides the user with the ability to formulate mathematical problems through code.

The library is available only in Python and also allows access to several different techniques and approaches to solve the problem posed, using syntax native to the language to facilitate and speed up the writing of code. Thus it is a library that serves as an interface to various tools or techniques for solving a problem, always using the same base code to formulate the problem.

Below, an example of the PuLP implementation of the first example posed in the Google OR-Tools section can be seen. A very clear similarity in the syntax can be observed.

Listing 3.4: Python implementation of a problem using PuLP

```
# Create the variables with values a domain of values [0, 1].
x = LpVariable("x", 0, 1, cat='Integer')
y = LpVariable("y", 0, 1, cat='Integer')

# Create a new problem.
prob = LpProblem("myProblem")

# Apply the constraint "variables sum must not be larger than one".
prob += x + y <= 1
# Apply the constraint "variable values must be equal".
prob += x == y

# Run the solver.
prob.solve()

# Obtain the value of variable 'x', in this case result will be 0.
x_value = value(x)
```

3.2 Web Application

A web application is an operating software that runs on the server rather than locally on the client's machine. Web applications are served to clients. Using the client-server model, clients can access the web application through web browsers with a network connection.

3.2.1 Context

Since the goal of the project is to help prepare for a trip, it is necessary to have an interface that allows the user to access the tools which provide functionalities to help the client solve certain sailing preparation problems, tools which are the goal of this project.

Since all the preparation takes place on land, it makes sense to use a web application instead of a mobile application.

3.2.2 Available Tools

In this section some comparisons between web frameworks that allow for the development of web application will be presented. Namely three open source tools: Django [10], Node.js [22] and Spring [30].

Table 3.1: Web Framework Comparison

	Django	Node.js	Spring
Language	Python. Which is a simple language to code with, from both the writer and readers perspective. One disadvantage is the fact it is not a typed language.	JavaScript. Used both on the client side and on the server side. Also not a typed language.	Java. Which is very verbose. Is a typed language. Kotlin [20] exists as an alternative, it is also typed, but features a much simpler language.
Documentation	Very vast, well written and clean documentation with a large community behind it.	Various resources and vast community.	Good quality documentation and vast community.
Development	Somewhat complex to configure, but development is quite fast thanks to using the python language and also thanks to Django's admin page and hot reload.	Simple configuration. Quick development speed thanks to the use of a single language on both server and client.	Although simple to configure, Spring itself is very complex as it provides a massive amount of options, as such, it requires vast knowledge of the framework.
Scalability	High.	Single Threaded, but still high thanks to background threads running tasks.	High.
Security	Besides being a very secure framework, it also provides authentication tools through a system that come directly incorporated with Django.	Not only is it less secure than other frameworks, but it also has some holes in its security.	Attention is required for for some security issues.

3.2.3 Django

Django is a framework used to handle database interaction, user authentication, API request processing, serves HTML templates provides a control panel that allows the user to directly interact and modify database objects, handling everything from database changes, table creation, migrations to even simple queries, inserts, deletes or updates. Turning the entire process of constructing and assembling the database from SQL to coding using the python language and the Django library.

The Django project also holds the python code used throughout the entire project.

After installing Django and the associated necessary components and finishing creating the project through the command line Django will generate a file structure that holds configuration files and other necessary files for the project to work.

Once the database is created and configured with Django, the project is already prepared to run. The authentication models are generated in the database without the need for user input.

File Structure

Other than the folder generated by Django that contains the project configuration. The user can use the command line to create "Django apps", these application are where the database table models, API views and admin page models will be placed.

Each app can be considered a category of the project and while all models are created in the same database regardless of category, it is useful on Django's side as a way to split the database into regions with little dependence on each other.

Inside the app the user can create a folder "models" or a file "models.py" in order to begin designing the database structure. Each Django model created in python inside this file will be created in the database once the user perform the migrate command, with the respective python fields transformed into their matching SQL fields.

Django generates an auto-increment "id" column to serve as the primary key for the model. However, the users can set a field's "unique" attribute to true in order to use that field as the domain-specific candidate key.

Listing 3.5: An example of a Django model in python

```python
# Import the model base class.
from django.db import models

# The model for an example.
class TestModelExample(models.Model):
    # A numeric attribute.
    attr = models.IntegerField(null=False, unique=True)
```

Listing 3.6: An excerpt from the SQL generated code for the respective model

```sql
-- Create the table.
CREATE TABLE public.djangoappexample_test_model_example (
    id bigint NOT NULL,
    example_numeric_field integer NOT NULL,
);

-- Set the id as the primary key.
ALTER TABLE ONLY public.djangoappexample_test_model_example
ADD CONSTRAINT djangoappexample_test_model_example_pkey
PRIMARY KEY (id);

-- Set the attribute as the domain-specific candidate key.
ALTER TABLE ONLY public.djangoappexample_test_model_example
ADD CONSTRAINT djangoappexample_test_model_example_attr_key
UNIQUE (attr);
```

Similarly, a respective admin model can be in an "admin.py" in order to allow the user to manipulate the database object through the admin panel.

Listing 3.7: An example of a Django admin model in python

```python
# Import the admin model base class.
from django.contrib import admin

# Register this admin model for the table "TestModelExample".
@admin.register(TestModelExample)
# The admin model class implementation.
class TestModelExampleAdmin(admin.ModelAdmin):
    # The columns to display in the admin page for the model.
    list_display = ['attr',]
```

Inside the "views.py" the user can create the API for the project in order to handle requests, be they simple JSON, or requests for full-blown web pages using templates.

Listing 3.8: An example of a Django views in python

```
# Class used for handling a specific API path.
class ExampleView(APIView):
    ...
    # The HTML template to be returned.
    template_name = 'path/to/html/template/page.html'

    # Handle the GET requests.
    def get(self, request):
        # Return the response, which will return an HTML template.
        return Response({
            # Data to be passed to the template.
            'some_key_accessible_in_templates': 'respective_value'
        })
```

HTML pages can be created using Django templating in order to populate them using data obtained during the request processing.

Listing 3.9: An example of how Django templating works.

```
// An HTML page.
<h1>
    // Placeholder which will contain the output data.
    {{ data.some_header_text }}
</h1>
```

Admin

As mentioned above the admin page allows the user to manipulate the database for tables whose models have a respective admin model created.

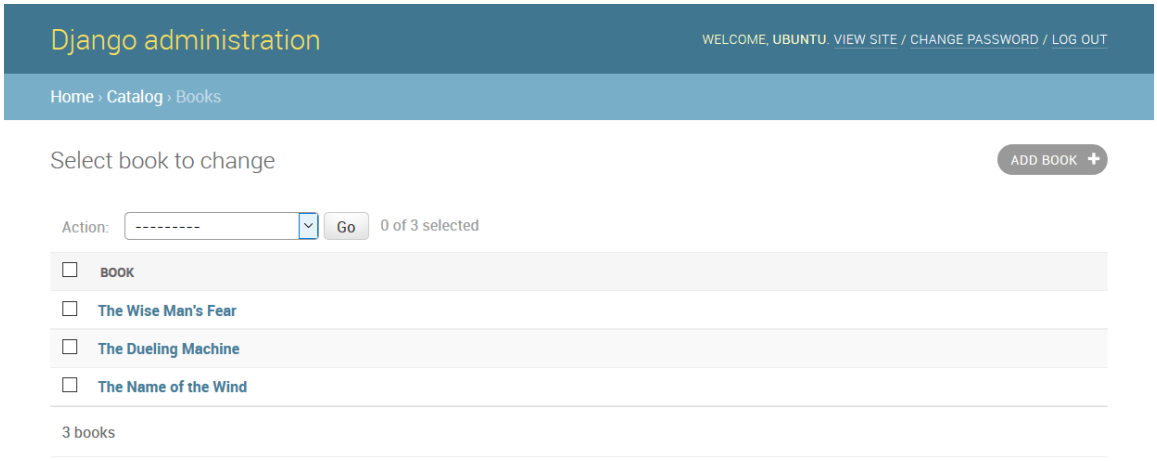


Figure 3.2: A Django admin page.

The page above is generated simply by having the user set which fields they wish to see listed.

Actions can also be added to the drop down visible on the image above that allow the user to execute python functions over the Django models which are reflected on the database.

This allows direct control, given the required permissions, for a user to view and manipulate database data without the need to construct any UI, as Django generates this views by using only the admin models. If needed, the user can also modify the HTML, JavaScript and CSS of the admin page to add additional features.

Authentication, Migrations and Logging

Django handles the authentication through the creation of database tables specifically focused on handling authorizations, tokens and logging.

It also has tables used for sessions, which is used for keeping track of user website login status. And finally, a table for migrations, which is used for keeping track of database level changes through models updates, for example, field changes.

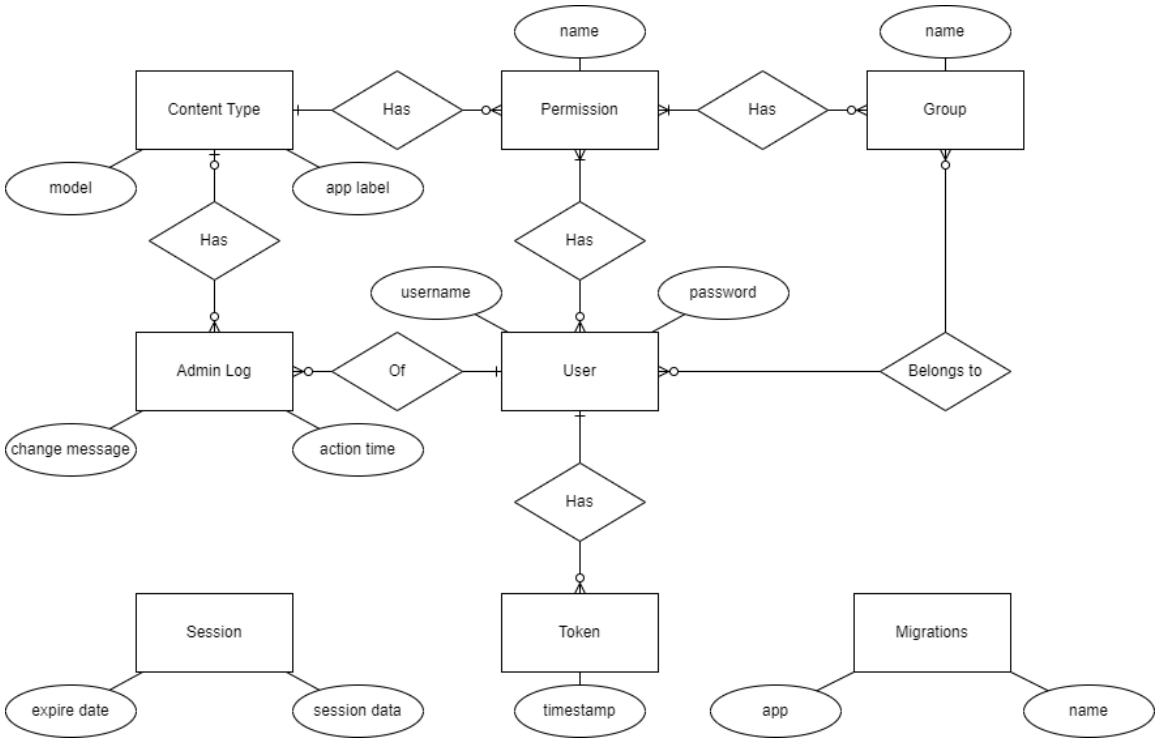


Figure 3.3: Django default tables

Above, it is shown the portion of the database that contains the tables necessary for the rest of the Django project to work. The tables are used for:

- **Authentication, logging and user sessions**, which allows for the existence of user accounts and the notion of session, which is used to maintain web client login state.
- **Permissions and groups**, are tables used for storing the user's access levels and their various permissions over the system and actions of Django's default admin interface.
- **Migrations**, which are used for updating the database table structure and keeping track of change history.

3.3 Mobile Application

A mobile application is software that runs on mobile devices. Mobile applications may or may not interact via APIs with an external server over a network with Internet access.

3.3.1 Context

It's important for users to be able to access information pertaining the trip, such as checking tasks or marking them as complete, consulting menus or shift information.

Although the preparation is done all on land, the actual execution of said preparation is done on the sea,as such it is also necessary for users to be able to easily access plan information during the trip. In this situation, the use of a mobile application is a possibility.

It should also be kept in mind that although a web application can be used on a smartphone, a mobile application has the advantage that it can be used without internet access. Being inside a ship far away from land, it is important to have the ability to work without a network connection.

3.3.2 Tools

In this section some comparisons between frameworks that allow for the development of mobile application will be presented. Namely frameworks with support for both Android [5] and iOS [16].

Some frameworks that support both of these platforms are, React Native [26], Ionic [15], Flutter [14] and Kotlin Multiplatform [21].

Table 3.2: Comparison of Mobile development frameworks

	React Native	Ionic	Flutter	Kotlin Multiplatform
Interface	More graphic interface flexibility while maintaining the native aspect.	Because of Webview, the process of designing mobile or web interfaces are the same, at the cost of the native look.	Allows for the usage of a single programming language for both platforms interface, sacrificing the native feel.	Very native feel thanks to native packages, but writing platform specific code is required for each platform.
Performance	Natively rendered, which increases performance.	Slower performance because of the use of Webview.	Requires the use of libraries to access native components, which lowers performance.	High performance, thanks to direct access to native components.
Documentation	Massive amount of documentation and large community.	Massive amount of documentation and large community.	Large support, but not as much as other frameworks.	As a recent framework, lower amount of support.
Development	Utilizes Javascript, a very common language. Quick Refresh function, that allows for synchronization of changes.	Can be written using TypeScript or regular JavaScript, which is a very common language.	Dart language, not that common. Quick Refresh. No need to write native code, only dart knowledge is necessary.	Kotlin allows for the writing of less (and more readable) code, with the large downside of requiring the writing of native code.
App size	Larger APK size with very high ceiling due to dependencies and libraries	Medium size APK files thanks to optimization	Larger APK size with very high ceiling due to dependencies and libraries	Small APK size, doesn't need libraries to use native methods

3.4 Translation

Because tools that provide either recipes, ingredients and products might not support the portuguese language, it might create some conflict between users or the tools themselves.

This is a problem because when trying to automate the menu creation and information extraction, we want the least amount of necessary user interaction either by the staff or the users themselves.

As an example, if we take recipe in english and want to search a portuguese supermarket website for a product which matches the one of the recipe's ingredient, the ingredient name must be translated from its original language into portuguese.

The python library "translators" provides an interface to multiple translation tools.

From commonly and freely available tools like google translate to other more advanced translation tools such as DeepL.

Listing 3.10: Code example of a sample usage of the translators tool.

```
# Import the library
import translators as ts
# Convert the text from english to portuguese, using Google Translate.
translated = ts.google("Hello", from_language='en', to_language='pt')
# Print the result.
print(f"Translated: {translated}")
# Output:
> Translated: Olá
```

4

Proposed Solution

This chapter details the solution used in order to solve the problems that were raised in previous chapters. It will detail the architecture and structure of the system as well as present the approaches to each specific issue as well as an overview of the entire solution.

The solution itself is constituted by a set of tools that are made available through a web application and a mobile application that are both used in order to make a plan and execute it, both applications communicate with a server that handles the actual plan making logic.

Although the project idea is to present a solution to the overall problem presented in the theme, it is better to frame the problem as an aggregate of different individual issues and thus approach each one with an individual dedicated solution.

The end result is an application that features various solutions to various issues. Below is a list of all solutions to be detailed and the respective problems they are addressing:

1. **Task Planning and Scheduling** - The approach taken to plan the schedule and its tasks.
2. **Meal and Product Planning** - The approach taken to handle menu design, ingredient weight and product shopping.
3. **Dish Ingredient/Product Extraction** - Address dish creation and respective ingredient product extraction.

4. **Database Structure** - Address the structure and solution to the problem of data persistence.

5. **Web Interface** - The middleman between user and the application logic. Which allows users to manage their plan data.

6. **Mobile Application** - Useful for keeping track of important data during the trip itself, where a network connection cannot be ensured.

7. **Deployment** - The logic used for deployment of the project.

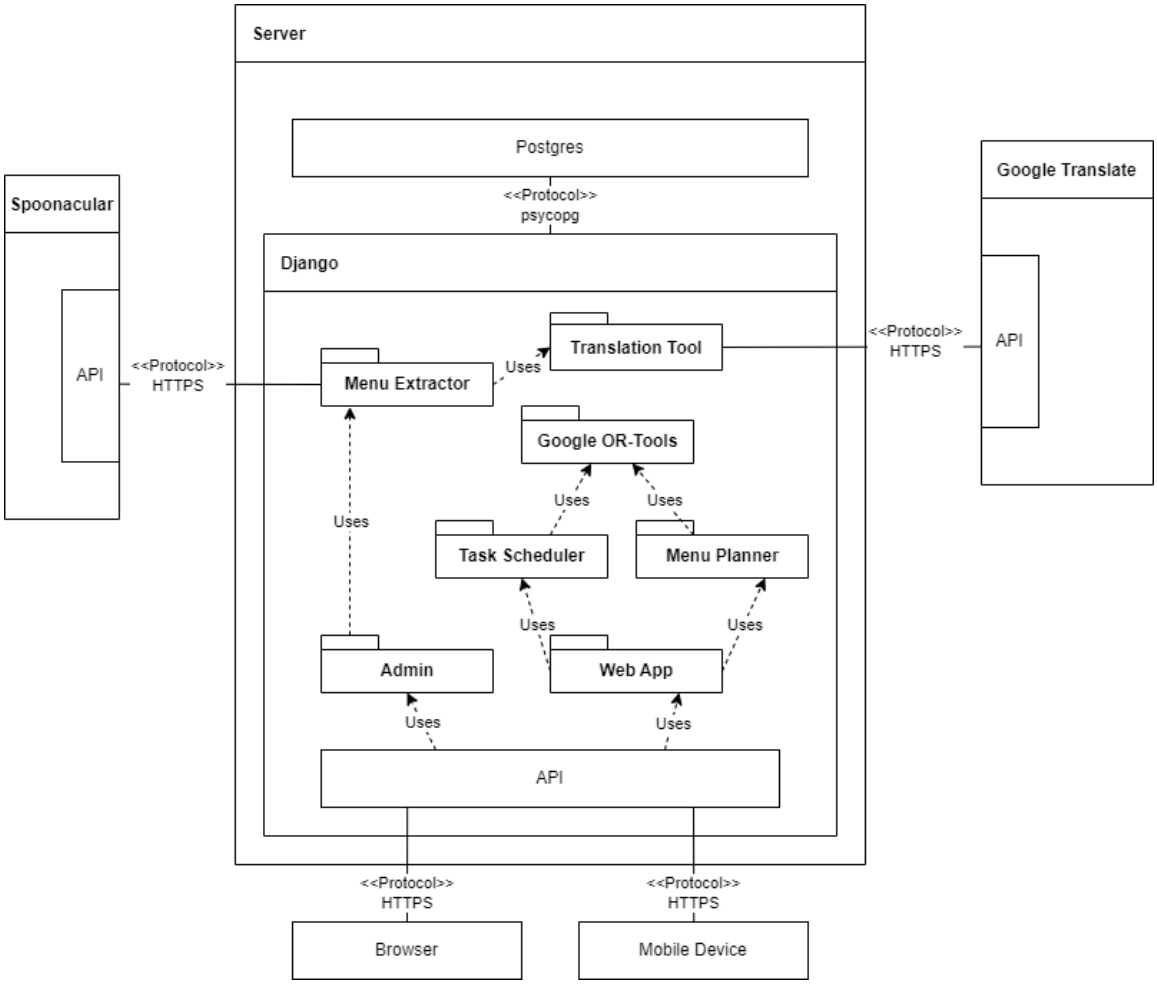


Figure 4.1: Diagram of the system structure

4.1 Task Planning and Scheduling

Google OR-Tools was the chosen tool in order to generate a schedule and to allocate the tasks between the crew members. The reason for choosing the Google OR-Tools is

that although it is not as flexible nor as simple to customize as other tools, it is faster, and the functionalities it does have are enough for the specific use cases in this project.

The solver used for this specific project is the OR-Tools' CP-SAT Solver. CP-SAT stands for a Constraint Programming (CP) solver that uses satisfiability (SAT) methods in order to address the proposed problem.

Although no article has been published on Google OR-Tools implementation [8], some information [9] was disclosed to give a better look into the solver.

The solver uses a lazy clause generation solver [29] on top of a SAT solver. All variables used on the solver are integers and the constraints are booleans resulting from comparisons between variables.

A pre-solving process is applied to the constraints until a fix point is reached. Once the fix point is reached, the solver can perform the search by asking the search heuristic to select an integer value and a direction for branching.[6]

This solver is mainly used for problems similar to the Employee Scheduling problem [12] and Job Shop problem [18].

For this project, the CP-SAT solver is used with the purpose of creating a task schedule, the steps consisted of identifying the variables, the states and finally the constraints.

4.1.1 Variable Structure

In order to create the variables it was necessary to create a tridimensional pseudo-array (using python dictionaries, commonly known as maps in other programming languages), the array's values are Google OR-Tools' custom boolean variables, "NewBoolVar", which are variables that will assume a value, either 0 or 1 when the solver begins its execution.

The dimensions of the pseudo-3d-array are as follow:

- **Task** - Each index number corresponds with a unique task present in the current sail plan. All tasks in the plan are ordered and each position corresponds with the index in the array.
- **Shift** - Each index number corresponds with a time slot, counting from the first shift, the day the shift occurs is not taken into account, for example, if there are 3 shifts per day, then index 0 is the first shift of the first day, and index 4 is the first shift of the second day.

- **Crew Member** - Each index number corresponds with a unique crew member participating in the current trip. All the participating crew members are ordered and each position corresponds with the index in the array.

There is also an auxiliary variable 2-dimensional array, it is the equivalent of the 3 dimensional array, except the third dimension "Crew Member" doesn't exist, and the value indicates solely if a specific task is to be executed in a specific shift.

This structure enables the problem to be written as code, we can represent each variable as:

```
# The expression bellow is the code representation of the statement:
# "task_x" is carried out during "shift_y" by "crew_member_z".
var_arr[task_x][shift_y][crew_member_z]

# Bellow is the code representation of the statement:
# Task 0 is carried out during shift 1 by crew member 2.
var_arr[0][1][2]

# The 2 dimensional array representing the statement:
# Task 0 is carried out during shift 1
var_arr_aux[0][1]
```

The value the variable takes, represents the truthfulness of the statement represented by that variable.

If the variable's value is 1, then that statement is true, if it's 0, then the statement is incorrect, either because it's the wrong task being executed in those circumstances, or because the shift where the task being executed is not the right one, or maybe the crew member executing the task is not the right person, or possibly everything about the sentence is wrong. A few more examples below:

- **The variable var_arr[x][y][z]** - Represents the truthfulness of the statement "Task x is executed during shift y by crew member z". As such, if the value of the variable is...
- **0**, then the statement is false.
- **1**, then the statement is true.
- **All variables in var_arr[x][y]** - This represents the list of all crew members possibly executing task x during shift y. As such if the value of each variable in this list is...

- **0**, then that means no crew member is assigned with executing task x during shift y, as such we can conclude that "Task x will not take place during shift y".
- **1**, then that means that all crew members are assigned with executing task x during shift y, as such we can conclude that "Task x will be executed during shift y by every possible crew member".
- **The variable var_arr_aux[x][y]** - This means that the task identified by the index number x, taking place in the shift identified by the index number y is valid combination and is meant to be executed as is. In short, "Task x will be executed by one or more people during shift y".

The truthfulness of these statements, combined with constraints, can be used to dictate the actions to follow, thus creating enabling the creation of a schedule.

The constraints are rules that restrict the solution generated by the solver, which means that any solution generated is guaranteed to have followed the rules applied to the model. If no valid solution exists (due to contradictory rules) then the solver will not output any solutions at all.

The constraints are represented in code by boolean expressions applied to the variables. For example, the expression:

```
varA == 1
```

Means that the solutions must have the value of the variable "varA" set to 1. Any solution where the variable "varA" takes the value 0 is invalid and thus will not be output from the solver.

It is also possible to create more complex logic, by using simple arithmetic operations in order to generate logic gates:

```
varA == 0 # NOT
varA + varB >= 1 # OR
varA + varB == 2 # AND
varA + varB == 1 # XOR
varA + varB < 2 # NAND
```

This logic can be applied to the problem at hand in order to create an expression representing an actual restriction that is relevant to the project. For example:

```
var_arr[0][0][0] + var_arr[0][0][1] + ... + var_arr[0][0][m] >= 1
```

The expression above is a representation of a possible rule: "At least 1 crew member must be executing the task identified by the index number 0 during the shift number

0.", or, translated into more user friendly terms, "At least 1 crew member must undertake this specific task during the first shift."

It is through the usage of these more complex expressions that it is possible to formulate any problem and its possible solutions, even more complex problems can be written using more than one expression.

For example, below is a single rule, constructed from multiple expressions:

```
# Rule fully written through all possible expression:
var_arr[0][0][0] + var_arr[0][0][1] + ... + var_arr[0][0][m] >= 2
var_arr[0][1][0] + var_arr[0][1][1] + ... + var_arr[0][1][m] >= 2
...
var_arr[0][n][0] + var_arr[0][n][1] + ... + var_arr[0][n][m] >= 2

# Rule written using a python loop.
# Iterate through each shift.
for y in range(0, n_shifts):
    # Create the sum for all people assigned to shift y.
    assigned_to_y = sum(actual_var for actual_var in var_arr[0][y])
    # Add the rule for each y (shift), number of people per shift >= 2.
    models.Add(assigned_to_y >= 2)
```

The set of expressions above represent a single rule: "The task identified by the index number 0, must be executed during every possible shift by at least 2 or more crew members."

4.1.2 Constraint Logic - Rules

As we've seen above, because the variable structures themselves already create a perfect representation of the problem, we can take advantage of this and can add the constraints.

Because the approach utilizes 2 arrays, one 3-dimensional and the other 2-dimensional, this can cause contradictions between each other, as such, we must also apply a special rule to ensure the result is consistent:

Listing 4.1: Bind the arrays, ensuring the variables assigned to one array won't contradict the other.

```
# Obtain the amount of people executing task X during shift Y.
task_x_shift_y_total = sum(var_arr[X][Y][0] + ... + var_arr[X][Y][Zn])

# Create the constraint, depending on the eventual
# value of var_arr_aux[X][Y] when the solver is executed.

# if var_arr_aux[X][Y] is true (task X occurs in shift Y)
#   then the sum of all people executing task X during shift Y
#   must be greater than 0.
(task_x_shift_y_total > 0).OnlyEnforceIf(var_arr_aux[X][Y])
# if var_arr_aux[X][Y] is false (task X does not occur in shift Y)
#   then the sum of all people executing task X during shift Y
#   must be equal to 0.
(task_x_shift_y_total == 0).OnlyEnforceIf(var_arr_aux[X][Y].Not())
```

Now that both arrays will be synced and no incompatibilities will appear, we can use the existing structure in order to create custom rules as options that are provided to the users to control.

- **Option#1** - The user inputs the shifts where a task can be assigned to.
- **Option#2** - The user inputs the people that are qualified to be assigned to a task.
- **Option#3** - The user inputs any other tasks that cannot be carried out by the same individual in the same shift.
- **Option#4** - The user inputs the minimum and maximum amount of individuals that can be assigned to a shift where the task can be executed.
- **Option#5** - The user inputs the amount of shifts that an individual executing this task must participate in sequence. Which is used for making an individual execute a single task multiple times slots in a row.
  - **Option#5.1** - The user may also indicate if they want rotations between shift changes of the same task.
- **Option#6** - The user can provide the X amount of shifts that they want the task to be executed in N shifts.
- **Option#7** - The user can provide the least X amount of shifts that they want every qualified person to be executing this task for in the space of N shifts.

Option#1 - Shifts where a task can be assigned to

The user inputs the shifts where a task can be assigned to.

This is an important option that allows the creation of tasks that represent objectives associated with a specific time of day. For example, it allows to generate tasks like "night watch" as you can simple only allow this task to occurring during the night.

Listing 4.2: Example of forcing a task not to be executed in specific shifts.

```
# Ensure that task x is not executed in shift y by making sure every
# individual (in the domain [0, Zn]) is unable to be assigned
# this task during the shift y.
sum(var_arr[x][y][0] + var_arr[x][y][1] ... + var_arr[x][y][Zn]) == 0
```

Option#2 - Which users are qualified to be assigned to which tasks

The user inputs the people that are qualified to be assigned to a task.

This option is useful as it allows for the creation of tasks that require certain individuals to do them. For example, the task "navigation" should only be done by someone who knows what they are doing.

Listing 4.3: Example of forcing a task to only be executed by a specific individual.

```
# Make it so task x cannot be done by individual 0 in shifts [0, Yn].
sum(var_arr[x][0][0] + var_arr[x][1][0] ... + var_arr[x][Yn][0]) == 0

# Skip individual 1 in order to ensure they can still execute this task.

# Make it so task x cannot be done by individual 2 in shifts [0, Yn].
sum(var_arr[x][0][2] + var_arr[x][1][2] ... + var_arr[x][Yn][2]) == 0

...

# Make it so task x cannot be done by individual Zn in shifts [0, Yn].
sum(var_arr[x][0][Zn] + var_arr[x][1][Zn] ... + var_arr[x][Yn][Zn]) == 0
```

Option#3 - Tasks that can't be simultaneously executed by a single user during a single shift

The user inputs any other tasks that cannot be carried out by the same individual in the same shift.

This allows for the execution of tasks like "daytime watch" that could conflict with "cooking" because if both tasks were assigned the same individual it wouldn't be possible for them to execute both tasks at once.

Listing 4.4: Example of forcing no overlap on task by the same individual.

```
# Make it so that individual 0 cannot do both tasks 0 and 1 on shift 0.
sum(var_arr[0][0][0] + var_arr[1][0][0]) <= 1
...
# Make it so that individual Zn cannot do both tasks 0 and 1 on shift 0.
sum(var_arr[0][0][Zn] + var_arr[1][0][Zn]) <= 1
# Make it so that individual 0 cannot do both tasks 0 and 1 on shift 1.
sum(var_arr[0][1][0] + var_arr[1][1][0]) <= 1
...
# Make it so that individual Zn cannot do both tasks 0 and 1 on shift Yn.
sum(var_arr[0][Yn][Zn] + var_arr[1][Yn][Zn]) <= 1
```

Option#4 - Minimum and Maximum crew members required for task execution

The user inputs the minimum and maximum amount of individuals that can be assigned to a shift where the task can be executed.

This allows for a variety of tasks. For example, it allows for tasks that can be assigned to any shift, but do not need to be, by making the minimum 0, there can be shifts that do not have this task being executed even if the shift itself allows for the task to be assigned to it. It also enables tasks that require multiple individual to execute them to exist, this by making the minimum value more than 1.

Listing 4.5: Example of forcing a task's shift to have a specific amount of people executing it.

```
# Make it so that the task x during shift 0 requires at least 2
# people to be present executing doing it.
sum(var_arr[x][0][0] + var_arr[x][0][1] ... + var_arr[x][0][Zn]) >= 2
...
# Make it so that the task x during shift Yn requires at least
# 2 people to be present executing it.
sum(var_arr[x][0][0] + var_arr[x][0][1] ... + var_arr[x][0][Zn]) >= 2
```

Option#5 - Number of tasks to be executed in sequence by a each individual

The input is the number of shifts an individual must execute this task in sequence. This is used for forcing an individual to execute a single task multiple time slots in a row.

This allows for tasks where the user wants to force an individual to do more than on shift in a row, for example, a user executing the task "night watch" should stay awake for multiple shifts in a row and then sleep for multiple shifts in a row.

Listing 4.6: Example of forcing sequential shifts for a user.

```
# Make sure either the previous or the next shift is also execution.
(sum(var_arr[x][y-1][z] + var_arr[x][y+1][z]) == 1)
    # Only if the task x is being executed in shift y by individual z.
    .OnlyEnforceIf(var_arr[x][y][z])
# Make sure the shift after the next doesn't have a task being executed.
(sum(var_arr[x][y-1][z] + var_arr[x][y+2][z]) == 0)
    # Only if the current and the next shift both have a task.
    .OnlyEnforceIf(var_arr[x][y][z], var_arr[x][y+1][z])
# Make sure the shift before the previous doesn't have a task being
    executed.
(sum(var_arr[x][y-2][z] + var_arr[x][y+1][z]) == 0)
    # Only if the current and the previous shift both have a task.
    .OnlyEnforceIf(var_arr[x][y][z], var_arr[x][y-1][z])
```

Option#5.1 - Tasks executed in sequence may want to diversify the crew members executing them

The user may also indicate if they want rotations between shift changes of the same task.

Still in the theme for the "night watch" task, the user may want to force at least some sort of rotations on sequential shifts, so that even if individuals have to do multiple shifts in a row, at least one person must be cycled out.

This allows for the creation of ladder shifts, where users A and B are on watch, once shift ends, B leaves and is replaced by C, at the next end A is replaced by D, next shift end replaced C by E and so on.

Listing 4.7: Example of forcing sequential shifts for a user.

```
# The code for this rule is complicated and difficult to
# explain even using pseudo-code so a simplified version is
# shown below for 2 sequential shift enforcement, also ignoring edges.

# Using exponents allows for the sum to represent a combination
# of who is and who isn't assigned to the task x in shift y.
individual_in_shift_y = sum(
    var_arr[x][y][0] * (2 ** 0) +
    var_arr[x][y][1] * (2 ** 1)  +
    ... +
    var_arr[x][y][Zn] * (2 ** Zn)
)
# Using exponents allows for the sum to represent a combination
# of who is and who isn't assigned to the task x in shift y+1.
individual_in_shift_y_1 = sum(
    var_arr[x][y+1][0] * (2 ** 0) +
    var_arr[x][y+1][1] * (2 ** 1)  +
    ... +
    var_arr[x][y+1][Zn] * (2 ** Zn)
)

# Ensure the shifts do not contain the exact same individuals
# on both sequential shifts.
(individual_in_shift_y != individual_in_shift_y_1)
```

Option#6 - Tasks that must be executed X times for in every N shifts

The user can provide the least X amount of shifts that they want every task to be executed in the space of N shifts.

This allows for flexible tasks that are freely to be executed as long as their execution isn't too far apart. For example, a task "daily logging" to keep track of the trip happenings, this task isn't critical, nor does it need to be executed at a specific time, as long as the task is done once a day it is fine. Thus this option can be used to ensure that, in the span of any spaced of 24 hours there is at least one task "logging" complete.

This is where the auxiliary array will be helpful.

Listing 4.8: Example of ensuring Y shifts of task X in K total shifts.

```
# The code for this rule is complicated and difficult to
# explain even using pseudo-code so a simplified version is
# shown below for 1 task x every 3 shifts.

# Obtain the total amount of task x executed for every 3 shifts
# starting from shift y. Note that var_arr_aux is being used
# instead of var_arr.
target = (var_arr_aux[x][y] + var_arr_aux[x][y+1] + var_arr_aux[x][y+2])

# Ensure at least one task occurs in interval starting from y.
sum(target) >= 1
```

Option#7 - Tasks that must be executed X times per each qualified user in every N shifts

The user can provide the least X amount of shifts that they want every qualified person to be executing this task for in the space of N shifts.

This allows for flexible tasks that are freely to be executed by an individual as long as their execution isn't too far apart. For example, using a combination of the option 3 (no overlap) and this option we can create a "negative task", which technically is just free time, because of forcing no overlap with any tasks, and thanks to rule 6 this free time may occur freely during any shift, but must occur for every individual at least once every Y shifts.

Listing 4.9: Example of ensuring an individual executes Y shifts of task X in K total shifts.

```
# The code for this rule is complicated and difficult to
# explain even using pseudo-code so a simplified version is
# shown below for 1 task x every 3 shifts for a single
# individual z.

# Ensure that every 3 shifts there is at least 1 shift where an individual
    z must execute the task.

sum(
    var_arr[x][y-1][z] +
    var_arr[x][y][z] +
    var_arr[x][y+1][z]
) >= 1
```

4.1.3 Scalability

Due to the existence of a three dimensional array populated by OR-Tools variables, any increase in the dimensions ("shifts", "crew members" or "tasks") will result in a large increase in the amount of variables the solver must handle, and thus slow both the time it takes for it to run and the memory occupied by the solver.

Figure 4.2: The graph shows the increase in the amount of time it takes to generate a schedule (y) according to the increase in size in all of the dimensions (x) used to generate a schedule.

Above we can see that the amount of time it takes to create a schedule increases massively with an increase over all the dimensions, with a total of 50 tasks, 50 crew members and 50 shifts, the solver started having issues generating a schedule due to both the amount of time it took and the memory used to create it.

While the large number of tasks and crew members are somewhat exaggerated and improbable, they aren't by any means impossible. And considering the fact the number of shifts is tied to both the number of shifts per day and the number of the days in a trip, the total number of shifts can indeed go much higher than the other dimensions. As an example, a total of 60 shifts could simply be only 5 days of trip with each day being split into 12 equal parts, and in this case, if instead of 5 days, it was 10, it would result in a total of 120 shifts.

One eventual solution would be to split the array in the "shifts" dimension in order to create multiple 3-dimensional arrays all with the same tasks and crew members, but

each with a portion of the shifts, and then run the solver over each section sequentially, while allowing the solver to "visit" the values obtained in previous sections in order to make sure the next generated section abides to the rules and options of the schedule.

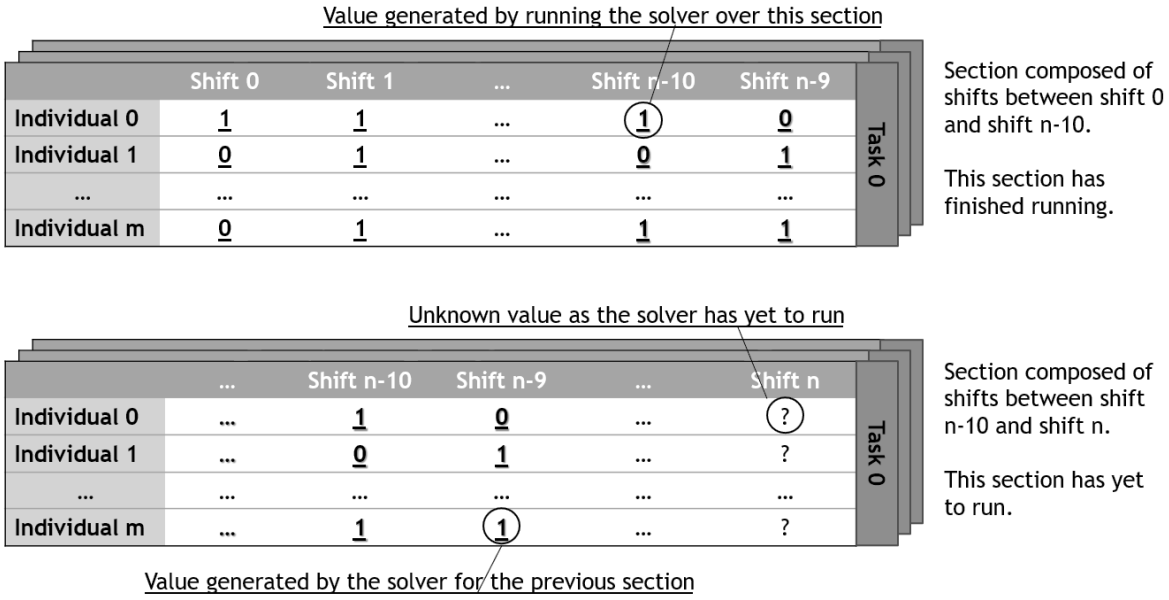


Figure 4.3: Illustration of the idea of splitting the shift dimension into multiple sections.

This is made possible due to the problematic dimension being the "shifts", had it been any other dimension this approach wouldn't be possible, as when the solver is determining the validity of the statement, "Crew member A performs task B during shift C", it takes into account every possible crew member and every single task, but only a small set of shifts, as such, as long as we provide the values generated by one solver to another, it is possible for the second solver to utilize the variable values generated by the previous solver in order to generate the next variable values.

4.2 Meal and Product Planning

Similar to task planning meal planning also make use of the Google OR-Tools library, but instead of using tasks and solely focusing on shifts and users, we instead focus on dishes and their ingredients, products and their information, and meal times and their meal type.

4.2.1 Information Structure

In order to construct the variables and the necessary constraints, it is required for the data to be structured in such a way that it allows for simple data lookup necessary for the operations that constitute the constraints.

- **Dish** - The dish related data is:
  - **ID** - A string which represents this dish.
  - **Types** - A list of strings that represents the types of meals this dish constitutes, if it's a small breakfast, a lunch course or any other type of meal.
  - **Ingredients** - A list of ingredients, where each ingredient contains the following data:
    - **ID** - A string which represents this ingredient.
    - **Weight** - The amount, in grams, of this ingredient that this dish requires for one individual.
- **Product** - The product related data is:
  - **ID** - A string which represents which ingredient this product contains.
  - **Price** - The cost of buying a single unit of this product.
  - **Weight** - The amount, in grams, of ingredient that one unit of this product contains.
- **Meal time** - The meal time data is:
  - **Index** - The meal time order number, similar logic to the shifts.
  - **Types** - A list of strings that represents the types of meals this meal time constitutes, if it's a small breakfast, a lunch course or any other type of meal.

Having established a structure for the information and relations between the different data, it is now possible to proceed to the creation of the structure for the Google OR-Tools variables.

4.2.2 Variable Structure

Because of how restrictive the operations between variables are, and in order to construct a simpler logic, three variable dictionaries were constructed:

- **Menu variables** - A 2-dimensional array, where the first dimension represents the meal time, and the second dimension represents the dish number, this structure is used to store boolean variables representing if a specific dish is the choice for a specific meal time. The idea is identical to the logic behind the shift scheduling.
  - Example, if the value of the variable, "menu_var[0][1]" is 1, it means that the dish represented by the number 1, will be the dish served during meal time 0 (which is the first meal of the trip).
- **Total menu variables** - An array, where the index represents the dish number, this structure is used to store integer variables representing the amount of times a specific dish used throughout all the meal times.
  - Example, if the value of the variable, "total_menu_var[3]" is 15, it means that the dish represented by the index number 3, will be served 15 times total during the trip.
- **Ingredient/Product variables** - An array, where the index represents the product number, this structure is used to store integer variables representing the amount of times a specific product is required to be bought.
  - Example, if the value of the variable, "total_product_var[7]" is 9, it means that the product represented by the index number 7, will be bought 9 times in preparation for the trip.

4.2.3 Constraint Logic - Rules

The library will assign to each variable of the model a random values, within the respective domain. This means that the model, by itself, cannot ensure that the ideas and purposes behind the creation of the variables and relations between those will be correctly displayed on the output.

That is why the use of constraints is required in order to turn all this logic into "rules" of the model's solver.

- **Rule#1** Determines that the sum of all menu/meal-time variable values for a certain menu is equal to the value of the total-menu variable.
- **Rule#2** Determines that the sum of all menu/meal-time variable values for a certain meal-time is equal to one. This, in order to ensure there can be no more than 1 menu served during a single meal time.
- **Rule#3** Determines that for each menu served, there must be enough ingredients, this is achieved by ensuring that the product-ingredient variable times the weight of each product's ingredient is equal or larger than the value of the total-menu variable times the ingredient needed for the menu, this for each ingredient of each served menu.
- **Rule#4** Determines that each menu/meal-time variable must have the value of zero, if that variable is the result of a combination between a menu classified as a certain meal type and a meal-time which does not allow for that specific meal type.

These rules are all implemented through the usage of the constraints supplied by the Google OR-Tools library.

Rule#1 - Sum of menu variables must be equal to total menu variable

In the case of the menu variables and total menus variables, the solver might fill every possible meal time of a specific dish with the value 0, indicating that during no meal time will this dish be served, all this whilst simultaneously assigning the value of 10 to the total menu variable of that dish, meaning that the dish is both never served at any meal, and also served 10 times throughout the trip.

In order to ensure this doesn't happen, we have to apply the following constraint, as shown in the pseudo-code below:

```
menu_var[X][0] + ... + menu_var[X][m-1] == total_menu_var[X]
```

In short, for every dish X, the sum of all variable values of every possible "menu variable" for each meal time of that dish must be equal to the "total menu variable" of dish X. An example can be seen below.

Table 4.1: Rule#1 - Example of a situation with 5 dishes and 3 meal times

	Shift 0	Shift 1	Shift 2	Total
Dish 0	menu_var[0][0] =0	menu_var[0][1] =1	menu_var[0][2] =1	total_menu_var[0] =0+1+1=2
Dish 1	menu_var[1][0] =0	menu_var[1][1] =0	menu_var[1][2] =1	total_menu_var[1] =0+0+1=1
Dish 2	menu_var[2][0] =1	menu_var[2][1] =0	menu_var[2][2] =1	total_menu_var[2] =1+0+1=2
Dish 3	menu_var[3][0] =0	menu_var[3][1] =0	menu_var[3][2] =0	total_menu_var[3] =0+0+0=0
Dish 4	menu_var[4][0] =1	menu_var[4][1] =1	menu_var[4][2] =1	total_menu_var[4] =1+1+1=3

Rule#2 - Only a single menu per meal time

To prevent the solver from attributing multiple menus to the same meal time, or from attributing no menus whatsoever to a meal time, a constraint must be applied to the menu variables.

```
menu_var[0][X] + ... + menu_var[n-1][X] == 1
```

In short, for every meal time X, the sum of all variable values of every possible "menu variable" for each dish of assigned to that meal time must be equal to one. An example can be seen below.

Table 4.3: Rule#2 - Example of a situation with 4 dishes and 3 meal times

	Shift 0	Shift 1	Shift 2
Dish 0	menu_var[0][0] =0	menu_var[0][1] =1	menu_var[0][2] =1
Dish 1	menu_var[1][0] =0	menu_var[1][1] =0	menu_var[1][2] =0
Dish 2	menu_var[2][0] =1	menu_var[2][1] =0	menu_var[2][2] =0
Dish 3	menu_var[3][0] =0	menu_var[3][1] =0	menu_var[3][2] =0

Rule#3 - Enough products must be acquired to make the meals

Because each menu requires ingredients to make, it is necessary to calculate the products and the amounts that will be needed in order to satisfy ingredient requirements.

That is why the construction of the information data structure was necessary. A menu requires a certain amount of ingredients and each product contains another different amount of ingredients.

We use the total menu variable in order to calculate the minimal amount of products that must be bought and use that minimum to force the total product variable's value domain to be at the very least the minimal amount to create the products.

```
# Obtain the needed amount, in grams, of ingredient X for each menu.
needed_amount_A = total_menu_var[A] * menu_data[A][X]['weight']
needed_amount_B = total_menu_var[B] * menu_data[B][X]['weight']
needed_amount_D = total_menu_var[D] * menu_data[D][X]['weight']
# Obtain the bought amount, in grams, of ingredients X.
bought_amount = product_data[X]['weight'] * total_product_var[X]
# Construct the constraint.
bought_amount >= (needed_amount_A + needed_amount_B + needed_amount_D)
```

Using this constraint ensures that no matter what combinations of menus are selected by the solver, the correct amount of each product that the crew will need to acquire will be indicated in the respective total_product_variable.

Rule#4 - Menus must be assigned to meal times of the same meal type.

Each menu can correspond to a specific type of meal, a menu that corresponds to a breakfast type dish cannot be served during a dinner. Each meal time can have multiple types, and each menu can also be assigned multiple types.

Table 4.5: Rule#4 - Example of a situation with 4 dishes and 3 meal times

	Shift 0 [Breakfast]	Shift 1 [Lunch]	Shift 2 [Dinner]
Dish 0 [Breakfast]	Menu 0's meal type is one of Shift 0's allowed meal types, as such, this menu can be assigned to this shift	menu_var[0][1]==0 Menu 0's meal type is not one of Shift 1's allowed types, as such, this menu CANNOT be assigned to this shift	menu_var[0][2]==0 Menu 0's meal type is not one of Shift 2's allowed types, as such, this menu CANNOT be assigned to this shift
Dish 1 [Lunch]	menu_var[1][0]==0 Menu 1's meal type is not one of Shift 0's allowed meal types, as such, this menu CANNOT be assigned to this shift	Menu 1's meal type is one of Shift 1's allowed meal types, as such, this menu can be assigned to this shift	menu_var[1][2]==0 Menu 1's meal type is not one of Shift 2's allowed meal types, as such, this menu CANNOT be assigned to this shift
Dish 2 [Lunch, Dinner]	menu_var[2][0]==0 Menu 2's meal type is not one of Shift 0's allowed meal types, as such, this menu CANNOT be assigned to this shift	Menu 2's meal type is one of Shift 1's allowed meal types, as such, this menu can be assigned to this shift	Menu 2's meal type is one of Shift 2's allowed meal types, as such, this menu can be assigned to this shift
Dish 3 [Dinner]	menu_var[3][0]==0 Menu 3's meal type is not one of Shift 0's allowed meal types, as such, this menu CANNOT be assigned to this shift	menu_var[3][1]==0 Menu 3's meal type is not one of Shift 1's allowed meal types, as such, this menu CANNOT be assigned to this shift	Menu 3's meal type is one of Shift 2's allowed meal types, as such, this menu can be assigned to this shift

```
# With X meal time and Y menu. If menu Y cannot be classified as
# at least one of possible types of menu allowed in meal time Y,
# then enter the IF statement and disallow any assignments of
# menu X to meal time Y.
if not any(
    one_of_menu_types in meal_time[X]['types']
    for
    one_of_menu_types in menu_data_info[Y]
):
    # Create the constraint.
    menu_var[X][Y] == 0
```

In short, the idea is that if the type of a menu does not match the meal types allowed in a meal time, a constraint is created that disallows the combination of that dish being served during that meal time.

4.3 Dish Ingredient/Product Extraction

The purpose of this process is allowing administrators to build up a catalog of a multitude of menus, containing both ingredient and respective product information for client usage in the meal planning section above.

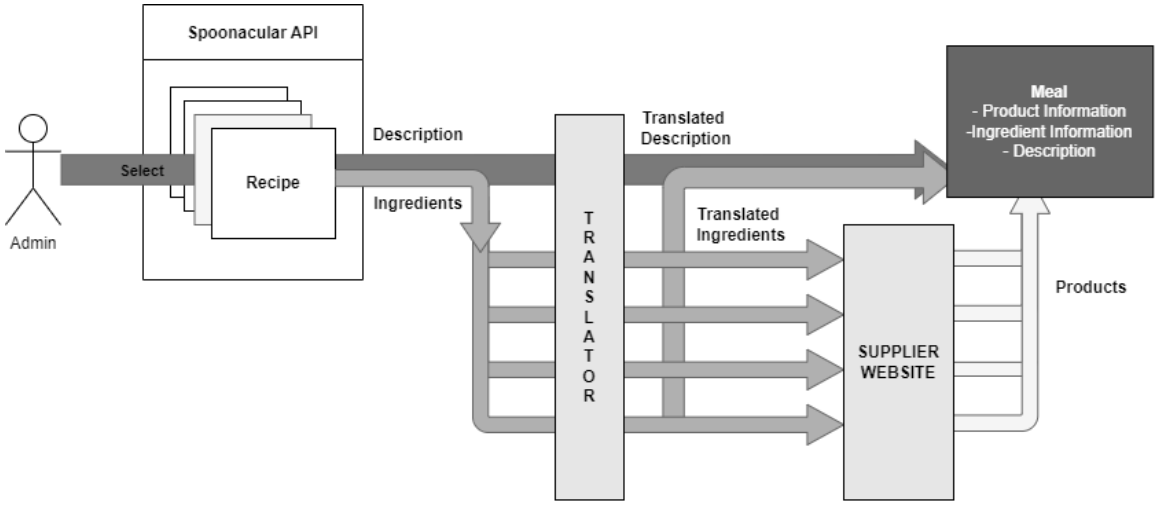


Figure 4.4: Process of meal menu construction

The process is mostly automated, with the only manual steps being the dish selection and the final verification to see if everything is in order and fix what isn't. The full sequence of actions that constitute the process can be divided into a few steps:

1. **(Manual)** The administrator inputs a dish into the tool.
2. Using the Spoonacular API, the input dish's information is extracted, such as the ingredient names, and their quantities.
3. Still using the Spoonacular API, the ingredients' quantities are converted to weight in metric system, specifically to grams.
4. Each ingredient is then translated from English to Portuguese.
5. Make an HTTP request to the Continente website's product search page and use the translated ingredient as the search query and retrieve the HTML content.
6. Parse the HTML and extract all product relevant information like price and weight.
7. Convert the weight into grams if needed.
8. Store the data in the database.
9. **(Manual)** Resolve any issues if they exists, for example, wrong product due to incorrect translation, no weight/price information or no product found.

4.3.1 Spoonacular API

This solution provides an API that allow for a multitude of food related operations while also possessing a massive database of recipes, ingredient, products, restaurant menus and other data.

It also allows for queries to be made in order to obtain recipes and ingredient information to create dishes, which are used in the project in order to build meal schedules.

The tool itself is free up to a certain number of "actions", with each action costing points, some more than others depending on the type of request and the amount and type of information returned. The amount of available points refresh monthly once expired for that month, the account owner must pay for more usage or simply wait until the month's end.

Some of the available actions, tools and features are such as but not limited to:

- **Recipe information**
- Meal price breakdown
- Restaurant menu item information
- Ingredient nutritional data
- Dish cooking procedures
- **Conversions between quantities**
- Advanced search filters for everything
- Food allergies and other health information
- ... and many more features

Above in bold are the tools that are directly useful to the project. While it does possess many features, for the scope of the current project, the usage of this tool will be solely focused in the process of menu creation. As such, the only necessary features that will be used are the recipe's ingredient information and conversion between an ingredient's quantities.

Listing 4.10: Example of the json data of an ingredient


```
...
{
    "id": 20081,
    "aisle": "Baking",
    "image": "flour.png",
    "consistency": "SOLID",
    "name": "flour",
    "nameClean": "wheat flour",
    "original": "2 tablespoons Flour",
    "originalName": "Flour",
    "amount": 2.0,
    "unit": "tablespoons",
    "meta": [],
    "measures": {
        "us": {
            "amount": 2.0,
            "unitShort": "Tbsps",
            "unitLong": "Tbsps"
        },
        "metric": {
            "amount": 2.0,
            "unitShort": "Tbsps",
            "unitLong": "Tbsps"
        }
    }
},
...
```


An example of the information on the ingredient "flour" obtained in the response from a random recipe request can be observed above. The json data features the necessary information and much more, and more can be obtained if needed, as the amount of information returned can be customized to include or exclude certain data.

For the current project, the important data is the ingredient's name and the amount.

And do note, that in the example above the quantity of the ingredient returned is not in grams nor in any unit that can be easily and directly converted to grams. This is often the case for liquids or any other ingredient that is usually used in small amounts.

As such, in the case of these types of ingredients, the API can be used in order to convert the quantity into grams in order for better to compute the total weight when designing the menu.

Listing 4.11: Example of the json data of an weight conversion

```
{
    "sourceAmount": 1.0,
    "sourceUnit": "l",
    "targetAmount": 1048.24,
    "targetUnit": "grams",
    "answer": "1 l orange juice are 1048.24 grams.",
    "type": "CONVERSION"
}
```

4.3.2 Translator Library

For the current project, the chosen translator engine was google translate mostly because it's fast and has no usage restrictions, while an engine like DeepL is vastly superior when it comes to more accurate translations, google translate suffices as the sentences being translated are a single word or just multi-words ingredient.

While it works for most products, the translation is not perfect, and will occasionally output an undesired result.

```
# Ingredient (EN) > Ingredient (PT) > Product (PT)

# Ideal result.
Juice of Lemon > sumo de (um) limão > limão
# Actual result.
Juice of Lemon > sumo de limão > sumo de limão
```

In the example above, when translating the ingredient "juice of lemon", which is the juice extracted by the simple action of squeezing a lemon, the resulting portuguese translation is: "Sumo de limão", which can mean both "juice of lemon" and "lemon juice".

Considering this string output will be used to query the supplier page for matching products, the returned product will not be "juice of lemon", as such a thing is neither common nor sought after, it will instead return results for "lemon juice", which is not an ingredient, but a drink. In fact, the ideal product should not even be "juice of lemon" either, but the actual lemon fruit.

4.4 Database Structure

The database management system (DBMS) adopted in this project was PostgreSQL, mostly due to its support and interaction with the Django framework as well as due to other useful features such as JSON field, which is used quite often in this project to store JSON data.

While the of mixing both relational and non-relational data might not be the best approach on some cases, as relational databases are optimized for relational data first, it is useful for the current project.

The project should be viewed as "tools and functionalities provided to the user through a web interface" than a "web interface that provides tools to the user". Which results in the focus of the project being more heavy on the scripts that allow the project to generate solutions that satisfy the given constraints.

For example, some usages of JSON fields in this project and their reasoning's are:

- **Storing the configurations of task options**, for usage when generating a schedule, these configurations/settings can change easily in terms of possible values and possible keys. Which, in the case of relational data, would result in needless updates to a massive amount of either old rows which wouldn't need updates or changes to the overall structure of tables, an example of this JSON is shown in listing 4.12.

- **External Product/meal data obtained through APIs**, this data, although parsed and used to fill respective tables, is also stored raw for the sake of validation, for both translations and extraction status. This is useful for debugging and logging of possible changes of external APIs.

- **Plan construction progress**, a plan is made up from various sections, some of which, don't necessary need to be followed in order. These sections are represented in the database by multiple tables. In order to avoid selecting from multiple tables through joins in order to verify the current progress, the usage of a single JSON field can be used to keep track and view the progress of a plan without using a single table JOIN.

Listing 4.12: Excerpt of the advanced configuration data of a task

```
{
    ...
    "valid_shifts": {
        "setting_type": "DATA",
        "setting_value": [0, 1, 2]
    },
    "sequential_settings": {
        "setting_type": "DATA",
        "setting_value": {
            "diversify_cast": false,
            "number_of_sequential_shifts": 2
        }
    },
    "people_per_shift_task": {
        "setting_type": "DATA",
        "setting_value": {
            "max_amount": 10,
            "min_amount": 0
        }
    },
    "per_person_cycle_settings": {
        "setting_type": "DATA",
        "setting_value": {
            "total_shifts_per_cycle": 6,
            "assigned_shifts_per_cycle": 2
        }
    }
    ...
}
```

Django serves as an interface between the programmer and the PostgreSQL database itself, handling everything from database changes, table creation, migrations to even simple queries, inserts, deletes or updates. Turning the entire process of constructing and assembling the database from SQL to coding using the python language and the Django library.

The figure 4.2 shows a diagram of the concept of the database structure used for this project, excluding all Django auto-generated tables.

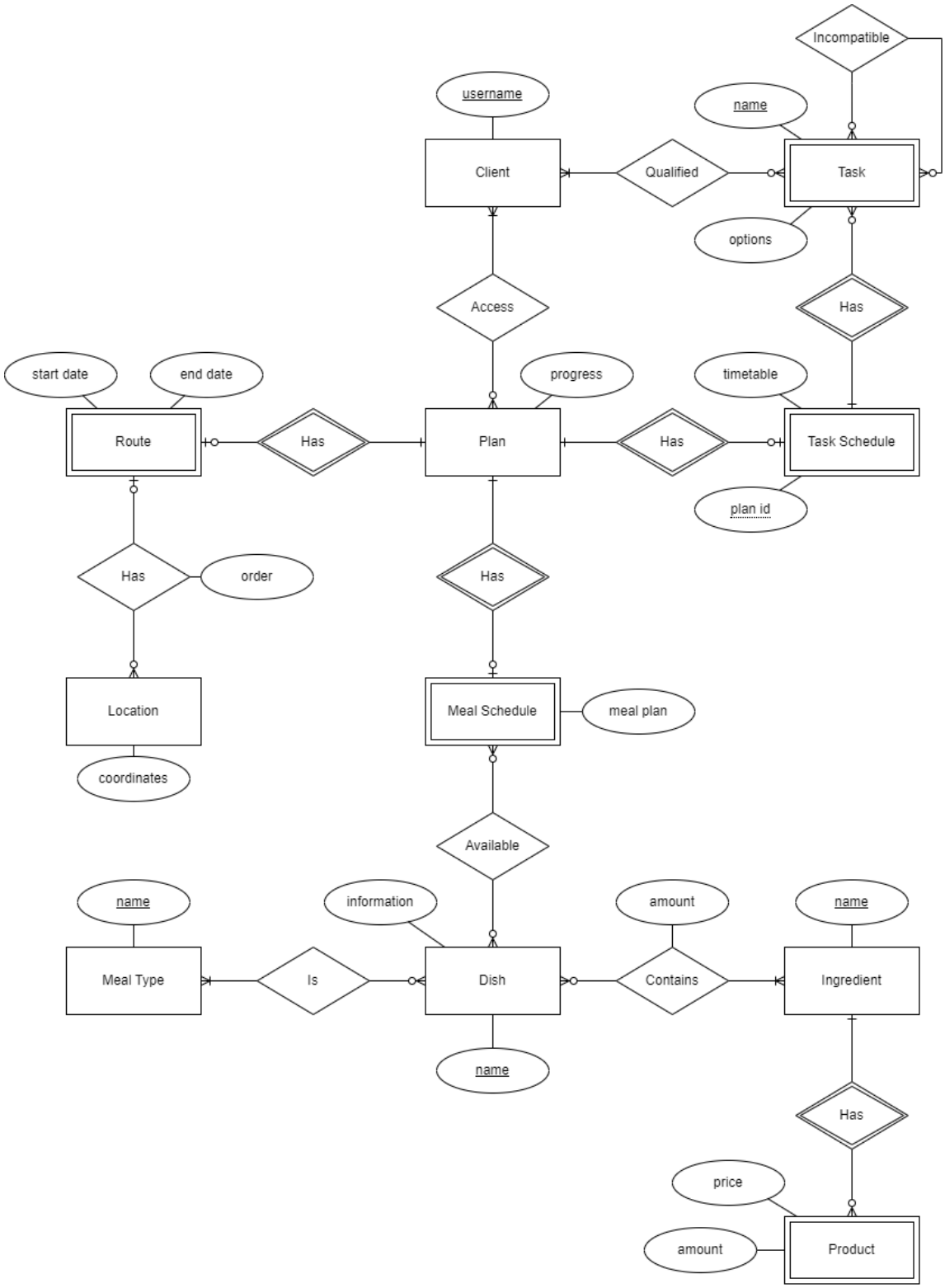


Figure 4.5: Concept of the database structure showing the most important entities, relations and attributes.

The database itself can be split into 5 categories:

- **System**, which are the Django generated tables and the Client table, they are used for authentication.
- **Plan**, which are the tables responsible for handling the general plan, both the plan table itself and the plan access resulting from the relation between the plan and the client table.
- **Route**, Simple tables with lower information, as they are not the focus of the project, contain data relation to the project start and end dates, and the route.
- **Nutrition**, tables related to the meal schedule, the meals themselves, the ingredients and respective dishes.
- **Tasks**, tables used for storing the tasks and the schedule generated from the tasks themselves.

4.4.1 System tables

The tables in this section are mostly Django generated. They handle authentication, sessions, logging and logic behind table updates.

Although Django also generates an "user" table, it is difficult to modify this table once created, as it mainly used for Django's inner logic.

As there is no reason to have two user tables, the original Django user was override before the creation of the table "user". This allowed for the creation of the custom user table: "Client".

The "Client" table stores user related information, for example, username. And is mostly referenced by Django's other generated tables.

4.4.2 Plan tables

The plan tables are used as the focal point of the database, as it's the "Plan" table that ties together all the steps (other tables) that make up a sail plan.

It contains information related to the plan as well as meta information relating to the plan progress and completion.

It also contain the table that holds the relation between a plan (Plan table) and the user (Client table), allowing for the concept of "access", which restricts viewing of the plan to a select number of allowed users.

4.4.3 Route tables

Tables holding less relevant logic for the project, such as the locations that make up the trajectory of the trip and the expected start and end dates.

This table was created in order to ensure that all the possibly useful trip data is kept, as to allow the creation of future features and tools that might need this data. Example: location of stores near stopping points.

4.4.4 Nutrition tables

This is one of more important focuses of the project, as such this section is larger than most others that make up the database.

Mainly containing data useful for the creation of a menu plan to be executed during the trip as well as other information extracted, in order to be used for fully documenting a dish and its contents.

This sections features a variety of tables:

- **Menu**, table containing the full menu information, such as the number of meals per day, the number of days where a meal will be consumed and the actual meal schedule.
- **Dish**, table containing dish information, mostly informative data such as the dish name, the cooking process and any other possible useful information related to the dish.
- **Ingredient**, this table is used for storing isolated ingredient information, as such, it solely contains the name of the ingredient.
- **Dish Ingredients**, this table contains the relation between each ingredient and a certain dish, such as the amount, original and in grams, of ingredient used in the dish.
- **Products**, information directly related to an ingredient's retail product, it contains information such as the weight and price of a product, which allows for optimizations about the overall menu.

4.4.5 Task tables

Tables related to the other main focus of the project, used to store data which allows for the generation and optimization of a schedule for task assignment and distribution of chores/activities/free time over a certain amount of users and shifts.

- **Schedule**, table which binds a certain number of tasks in order to generate a schedule which distributes each tasks according to its respective rules and collective rules.
- **Task**, table containing the full task information, such as the more technical task options, the description and any other assignment related information.
- **Qualified Users**, table representing a relation between a client and a task, used to define which users are qualified to complete a certain task.
- **Incompatible Tasks**, the table defining the self reference of tasks that cannot be executed simultaneously by the same individual during a shift.

4.5 Web Interface

The web UI is simple and is split into a small number of pages with different functions.

The pages are constructed using the Django templates, which allow for the writing of HTML, CSS and JavaScript while utilizing context variables that are generated on the response.

The theme, both the HTML and the CSS, used for this project was adapted from the Jango theme, which is one of the freely available themes provided by KeenThemes [19].

4.5.1 Home Page

The home page has a simple layout, allowing authenticated users to navigate to other pages and providing non-authenticated with a landing page from which to perform the login or registration.

4.5.2 Register and Login

Both the login and registration page are simple forms where the user can perform the registration, or login if they already have an account.

The login page also features a remember me button which persist the session beyond closing the browser using Django sessions.

4.5.3 Plan list

The plan listings page displays a grid of both the plans the user participates in, and other public plans available for viewing.

Search fields and other filtering options are available for easier finding of specific plans.

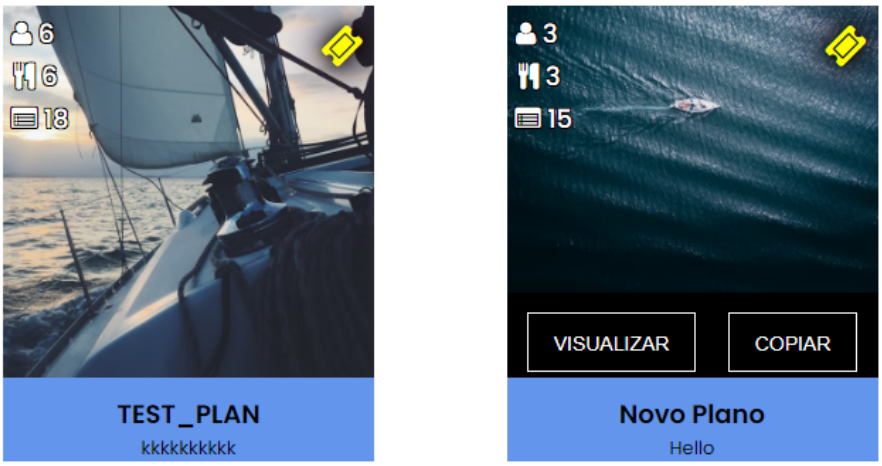


Figure 4.6: Two plans, the right one hovered by the cursor.

In the image above, two items of the grid are visible, each one representing a different plan. The plan name and description is below the image featured in each item.

There are also 3 items stacked vertically in the top left of either plan. Each item represents a different information:

- The topmost icon and number indicates the size of the crew, this makes it easier for other users to find a plan that matches their circumstances.
- The middle icon indicates the total amount of meals that the plan has. Which is helpful information combined with the information on the last icon.
- The bottom icon is the total amount of "shifts" throughout the trip. For example, a trip that lasts 5 days and has 6 shifts a day will result in a total of 30 shifts.

On the top right of both plans, a yellow ticket is visible. The yellow ticket is an indication that the user takes part in that specific plan, if a user does not belong to a plan, then no icon will yellow show up.

On the plan on the right, two buttons can be seen. On hover, the image will slide up slightly and reveal a small menu that allows users to navigate to a plan or create a copy of that plan.

The list only displays public plans or plans the user takes part in. Private plans or plans that have yet to go public are not shown to the user nor are they available to visualize or copy.

4.5.4 View/Change Plan

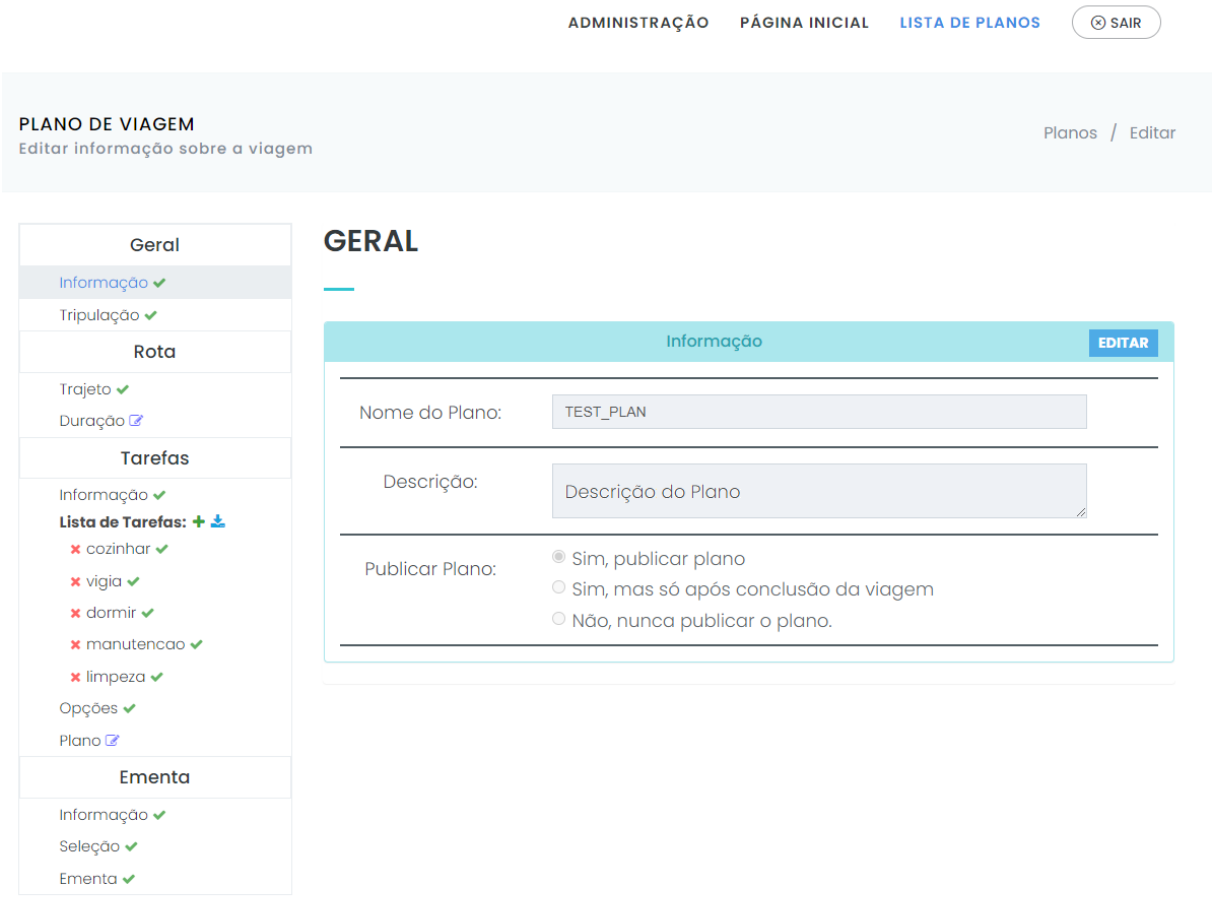


Figure 4.7: The web page featuring the plan information

The image above shows the structure of the web page used to view and modify a plan. The top of this page is the same as the previous pages, featuring both the header and the banner with information related to this page.

Below that section, there are two key things elements visible. The sidebar on the left, and the form with text inputs and options inside the panel named "Informação" below

the large "Geral" text.

Figure 4.8: The plan's progress tracker and completion.

The sidebar contains the sections that make up a plan and each section's progress. Clicking on a section will change the panel on the left to that section.

Each section has one of 3 states, locked, in progress, and completed, respectively indicated by the lock symbol, the edit symbol and the check mark symbol.

Some sections depend on other's completion and thus are only unlocked when the necessary sections are finished. For example, in the picture above, each task depends on the overall information provided by the user and thus they can only be accessed

when the "Informação" section, where the general task information is completed.

Each section works the same way, once the user clicks it, the panel on the right will change in order for the user to input the necessary information. The only exception is the task sections, they function the same way as every other section with the difference being that tasks can be added, imported or removed by any user with plan editing privileges.

Figure 4.9: The plan's information section panel.

In the picture above, the panel for a section can be observed. All the inputs within the panel are disabled until the "edit" button on the top right is pressed. Once pressed, the button will change into a "save" button. Once a section is successfully saved, it will be marked with a check mark on the left side menu and other locked sections will unlock for a user to complete them.

If the user does not have permission to edit the plan, the edit button will not show up and the user will be unable to make any changes to the plan.

4.6 Mobile Application

The mobile application was created using Flutter. It is a small and simple application that only gives the user two functionalities, managing their own tasks and viewing the menu meals.

The idea of the app is to allow users to view the relevant trip information without having network access during the trip itself.

4.6.1 Login

The login page is simple and uses "shared preferences", which are key-value pairs used to locally store data. It is used to store the token received from the server after the login.

If no token is stored when the app opens, the login screen will be launched.

4.6.2 Plan Selection

The plan selection page lists various plans for users to select, only plans that are finished and ready for usage are displayed. Plans that are still in the process of being made do not show up on this screen.

4.6.3 Task Schedule - Left Tab

Figure 4.10: The page featuring a timetable containing every task (left) and the menu that displays when a task is clicked on the timetable (right)

This screen is shown when the left tab on the bottom of the screen is selected. It displays a timetable with every task this user must execute being shown in the table.

The user can check the task information by tapping the respective task in the time table, which opens a popup menu that display not only the task information but also the current state of the task, which the user can mark as "in progress" or "completed" to keep track of tasks.

4.6.4 Meal Information - Right Tab

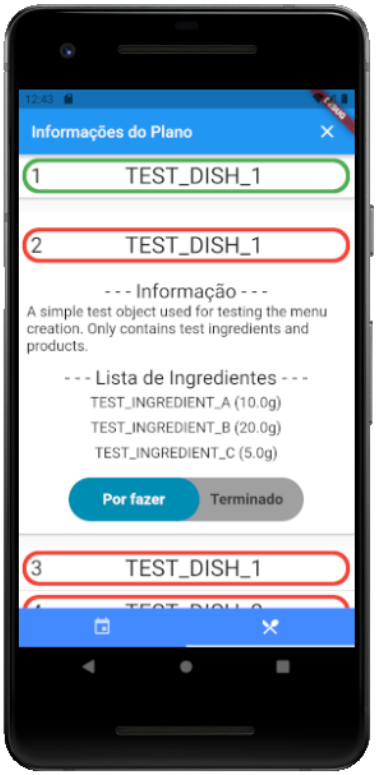


Figure 4.11: This screen displays the list of meals throughout the trip, further details can be viewed by expanding each item.

The screen above is shown when the right tab at the bottom of the screen is selected. It displays a list of all meal times throughout the trip, each meal time has a menu associated with it, upon clicking one the menus in this list additional information will be shown about the menu such as the required ingredients and their amount as well as a description that describes the cooking process or any other relevant information.

Each list element, when expanded will also display a switch that allows the user to mark the meal preparation as finished. This allows for the user to keep track of the prepared meals and also enables the user to keep track of any meals that have been prepared in advance in the case of a menu being made earlier.

4.7 Deployment

The Django project was implemented in an Ubuntu environment (18.04/20.04 LTS) with a local database.

Thanks to how Django is set up, the application can run in other system, even those that do not support the database itself as Django is able to work with a remote database with minimal effort.

For this specific project however, the choice of using the same Ubuntu system is the most fitting as not only is the environment open source as it's easy to setup for both development on a virtual machine as well as a server without worrying about licenses for software.

While the Django project is builtin with capabilities to serve itself in a server, it's not a feature that is meant to be used alone for deployment in a serious production environment. That is why Django is only one of three blocks used for the actual production.

The second block is Gunicorn, which will serve the Django project. Gunicorn is a python HTTP server designed to server client on low latency and fast connections.

Although nothing else is needed to run a server, Gunicorn is not meant to handle slow, buffering inducing clients and thus is susceptible to denial-of-service attacks. Due to this, Gunicorn will instead act as a middleman between the project and the web server.

The web server is the final block of this 3 brick layer. The chosen web server was Nginx, which is the tool used serving static content and acting as a reverse-proxy, which will communicate with the outside and listen to internet requests and direct them towards Gunicorn.

With this structure in place the Django project can much more safely begin to receive request and process them while behind Gunicorn and Nginx. And the server is now ready for production.

5

Conclusion

The finished project gives those that wish to plan a sailing trip to do so without the needs to get stuck on logistics, so they may focus on the the more nautical aspects needed when sailing.

They can dedicate more time to consider important parts of the planning such as the nautical route, weather, tide considerations, the equipment and navigation, instead of worrying and wasting time on discussions that could be avoided, the "what to buy for the trip?", the "where to buy it?", "who is going to do what tasks?", "what meals are being made", "who's making them?", "who's keeping track of what's done and what isn't", "is this enough ingredients", "do we need more?", "what about the total weight, is this too much for the boat? who's even counting?".

The research conducted revealed the lack of solutions for the overall problem, as most of them are focused on more specific, individual situations, the implemented solution addresses most of these situations in the context of the problem.

The ability to create schedules within a minute, create tasks to fit your idea of a plan and share them with others, as well as just using tasks made by others. To generate an entire plan within seconds. To allow multiple people the ability to all work on the same "drawing board". To keep track on what's done and what's still to do.

It's an application that is solely focuses on the logistics side, which frees up the necessary time to focus on other essential components of trip planning.

Future Work

The objective of the project is to provide tools to assist on the logistics side of sailing. As such, the project can never truly become finished in the truest sense of the word, as it can always keep expanding and more features and capabilities can be added.

Some possible adjustments are the improvement of user experience and additional focus on the web and mobile aspects and experience, which currently are to the project, the "window" to the actual content as such they lack the sophistication and visual aspect of modern websites and applications.

Also features that improve user friendliness and focus on both providing advanced users with tools to customize their plan to the maximum while also maintaining the idea of allowing less advanced users to construct their sail plans without much hassle. Keeping the spirit of sharing the plans and allowing the users themselves to create a self improving platform.

Other features that can bring the application forwards are, the ability to view nearby markets on the sail trip stop points, the ability for the application itself take the stops into account to split the products needed to be bought at each stop to reduce the overall ship load.

The application itself, because of its focus on the logistics side specifically, can also to become a "half" for another project that focuses on the more nautical aspects of the trip itself, resulting in a project that oversees every aspect of the sailing plan.

Or even go beyond sailing itself and expand onto other fields, the projects features could easily be applied as solutions to multiple other problems.

References

[1] Bob Bond, *The Handbook of Sailing*. Knopf, 1992.

[2] Steve Sleight, *The Complete Sailing Manual*. DK Publishing, 2021.

[3] David Seidman, *The Complete Sailor, Second Edition*. McGraw-Hill Education, 2011.

[4] Salido M.A. Rossi F. Barták R., "Constraint satisfaction techniques in planning and scheduling", *J Intell Manuf*, vol. 21, no. 1, 5–15, 2010.

[5] Open Handset Alliance Andy Rubin ... Google Baidu. "Android". (Sep. 2008), [Online]. Available: https://www.android.com/.

[6] Laurent Perron and Frédéric Didier. "Cpaior 2020 master class: Constraint programming". (), [Online]. Available: https://www.youtube.com/watch?v=lmy1ddn4cyw.

[7] Standard C++ Foundation. "C++". (1985), [Online]. Available: https://isocpp.org/.

[8] Laurent Perron. "Solver clarifications 1". (), [Online]. Available: https://github.com/google/or-tools/issues/920#issuecomment-435880431.

[9] ——, "Solver clarifications 2". (), [Online]. Available: https://stackoverflow.com/questions/57123397/which-solver-do-googles-or-tools-modules-for-csp-and-vrp-use/57125734#57125734.

[10] Django Software Foundation. "Django". (Mar. 2022), [Online]. Available: https://www.djangoproject.com/.

[11] Microsoft Corporation. ".net". (2002), [Online]. Available: https://dotnet.microsoft.com/.

[12] Google OR-Tools Documentation. "Employee scheduling". (), [Online]. Available: https://developers.google.com/optimization/scheduling/employee_scheduling.

[13] Jeremy Waters. "Fastseas". (), [Online]. Available: https://fastseas.com/.

[14] Google and community. "Flutter". (May 2017), [Online]. Available: https://flutter.dev/.

[15] Drifty. "Ionic". (2013), [Online]. Available: https://ionicframework.com/.

[16] Apple. "Ios". (Jun. 2007), [Online]. Available: https://www.apple.com/ios.

[17] Oracle. "Java". (1995), [Online]. Available: https://www.java.com/.

[18] Google OR-Tools Documentation. "Job shop". (), [Online]. Available: https://developers.google.com/optimization/scheduling/job_shop.

[19] "Keenthemes". (), [Online]. Available: https://keenthemes.com/.

[20] JetBrains. "Kotlin". (Jul. 2011), [Online]. Available: https://kotlinlang.org/.

[21] ——, "Kotlin multiplatform". (), [Online]. Available: https://kotlinlang.org/lp/mobile/.

[22] OpenJS Foundation. "Node.js". (Mar. 2022), [Online]. Available: https://nodejs.org/en/.

[23] Google. "Google or-tools". (Mar. 2022), [Online]. Available: https://developers.google.com/optimization.

[24] Stuart Mitchell, Michael J. O'Sullivan, and Iain Dunning, "Pulp: A linear programming toolkit for python", 2011.

[25] Python Software Foundation. "Python". (1991), [Online]. Available: https://www.python.org/.

[26] Facebook and community. "React native". (Mar. 2015), [Online]. Available: https://reactnative.dev/.

[27] "Sailing europe". (), [Online]. Available: https://www.sailingeurope.com/.

[28] David R Karger, Cliff Stein, and Joel Wein, "Scheduling algorithms.", 1999.

[29] Peter Stuckey. "Search is dead". (), [Online]. Available: https://people.eng.unimelb.edu.au/pstuckey/PPDP2013.pdf.

[30] VMware. "Spring". (Mar. 2022), [Online]. Available: https://spring.io/.

[31] "Sunsail". (), [Online]. Available: https://www.sunsail.com/.

[32] "Windy.app". (), [Online]. Available: https://windy.app/routing.