Project: Find Me A Parking Space!

As many other cities around the world, Tartu has not been designed with a layout to hold large parking supplies. It is therefore important to design policies that serve to find a balance to encourage a smooth traffic and secure conditions for pedestrians, particularly in the historical center of the city. Aligned with this vision, the city government has engaged your team for implementing a system for managing the parking spaces in Tartu.

General information

- The project has to be completed in teams of 4 students (register your team here).
- The project has three four checkpoints (assessments) on 17.11, 24.11, 01.12, and 08.12.
- Select a time slot for the checkpoint using this Doodle.
- The code with the solution must be pushed into a private repository (only one repository per team). You can use GitHub, Bitbucket, or GitLab, as you prefer.
  - Please add all the teachers to your repo -- the emails are posted on the home page of the course website.
  - If you use Bitbucket, make sure you didn’t reach the user limit. In case you have reached the limit, you can upgrade your plan by using your UT email account.
- You must provide a solution based on the requirements described in this narrative. If you are not sure about a requirement, please ask your teaching assistant for clarification.
- The artefacts generated (e.g. commit logs, documentation, issue tracker) should show clear evidence of the participation of all team members.

Description

Paid parking in Tartu concerns only a small zone around town center. Clearly, that zone is the one that shows a large concentration of pedestrians (e.g. tourists and University students) and is also the one with important space restrictions. In fact, we will take the parking zones currently used by the city government, defining the street segments for on-street and off-street parking and the corresponding fees. Note that off-street parking provided by the City government is free but time restrictions may apply. On the other hand, on-street parking is always paid.

Figure 1 shows a map of Tartu and its parking zones. Tartu city government provides zones for free parking without time limit, free with time limits (maximum of 60 and 120 minutes), and two paid zones (zone A is highlighted in red and zone B in blue). Up to now, paid parking in Tartu uses an hourly fee scheme: the total amount to be paid must be calculated based on the number of started hours that the car stays in the parking space. However, we would like to add a new policy to provide a “real-time parking fee”: the amount to be paid would be computed using a fixed fee per 5 minutes period. Table 1 shows the fees.

<table>
<thead>
<tr>
<th>Zone</th>
<th>Payment type</th>
<th>Hourly (euros per hour)</th>
<th>Real-time (cents per 5 minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1 - Parking fees.
The Minimum Marketable Product (MMP) of the system must contain the following features:

User Management
- The system should allow users to self-register, log in, and log out. Each user must be saved with their basic details such as name, email, and license number.

Interactive search of parking space
- When you enter a destination address or point of interest (e.g. “Narva mnt 18” or “DELTA”), the system must present a summary of available parking spaces around that address. The information about the parking space availability must be presented, which includes information about the price that applies (Zone A or B) and the distance from the destination address to the available parking space.
- If the user additionally enters the intended leaving hour, the system must also provide an estimation of the fee to be paid (per zone, hourly vs. real-time payment)

Parking payment
- The system must allow a car driver to select between hourly or real-time payment
- The system must allow a car driver to submit a start and end of parking time. Using the geolocation capabilities, the system must identify the parking space been used by the car. The backend must then block the corresponding parking space and update the availability of such parking space (this implies that the information displayed in the client-side must be updated accordingly).
If the car driver selected an hourly payment scheme, the system must notify the driver 10 minutes before the end of the period already paid. If the driver does not extend the period, the system would start advertising the parking space 2 minutes before the end of the paid period.

Billing/Invoicing
- The system must implement a payment mechanism. For instance, you could provide a pre-paid system where each user has an account with a certain amount of money and the payments made by that user debit the corresponding amount of money from the account.
- The system must generate and record an invoice for each payment.
- The user (car driver) may pay using one of the following options:
  - Just before starting a parking period or when extending the parking period, when hourly-based payment is selected
  - At the end of a parking stay, when real-time payment is selected
  - At the end of each month (the application would provide then a configuration section, where the customer could configure this option), when real-time payment is selected

Your task
You must implement a web application using Phoenix that serves as a parking management system and fulfills the requirements of the MMP described above.

To achieve the quality required of the MMP, we ask you to implement the following agile practices:

- **TDD.** As part of the TDD cycle, you must write unit and/or integration tests that verify the logic of your solution. You may include several assertions in a single test or write several test cases; both approaches will be fine as long as the cases described in the scenario are covered.

- **BDD.** As part of the BDD cycle, you must provide Gherkin user stories and white bread steps that are in line with the scenario and implement the corresponding acceptance tests.

- **CI/CD.** You should implement a branch strategy and enable a CI/CD mechanism. You can use the tools and environment you prefer for this task. We can recommend an environment that consist of Bitbucket, Jira, Bitbucket Pipelines, and Heroku. Here are some useful links.
  - [https://github.com/semaphoreci-demos/semaphore-demo-elixir-phoenix](https://github.com/semaphoreci-demos/semaphore-demo-elixir-phoenix)
  - Takso configuration file example for Bitbucket pipelines [https://bitbucket.org/ezequielscott1/asdexample/src/master/bitbucket-pipelines.yml](https://bitbucket.org/ezequielscott1/asdexample/src/master/bitbucket-pipelines.yml)

- **Scrum.** You should follow Scrum as the framework that supports the management of your project. You should provide evidence of the execution of all the practices you implement. You can apply variations of the framework, but those variations should be clearly justified. We recommend you using this checklist:

In addition, please commit the code to your repository as often as you can and use informative commit messages.
Grading

In total, you can get a maximum of 32 points for this project. Team members receive equal grades.

In general, the grade is determined based on:

- The compliance with agile practices, as they were learned during the course
- The scope of the solution (how much was implemented?)
- The functional correctness of the solution (does it work?)
- Functional validation

The project and its assessment will be split into four 1-week sprints. At the end of each sprint, teams will receive marks based on their progress and the execution of agile practices. Each sprint has a goal that teams must achieve (see Table 2) in order to get maximum points (8 points each).

Table 2 - Sprint details.

<table>
<thead>
<tr>
<th>Sprint</th>
<th>Checkpoint date</th>
<th>Goal</th>
<th>Max points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.11</td>
<td>The goal of this sprint to set-up the process, the environment/tools needed, and the product and sprint backlogs in order to be ready for development. During the checkpoint, you should show the scrum checklist indicating the practices you will implement, the development environment ready, and the product and sprint backlogs with the corresponding user stories.</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>24.11</td>
<td>The goal of these sprints is to implement/test the user stories that were allocated to the sprint backlogs.</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>01.12</td>
<td>During the checkpoints, the team should be able to show at least 2 user stories fully implemented and tested.</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>08.12</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

Following this schema, each team is free to decide what functionalities to implement on each Sprint according to their own plan.

Right after Sprint 4, we will assess the whole project by checking the completeness of the solution, including both implementation and testing. As a result, the total marks will be adjusted according to the following calculation:

\[
\text{Project} = \text{Sprint1} + (\text{Sprint2 + Sprint3 + Sprint4}) \times \text{Completeness}
\]

Where Completeness is a factor between 0 and 1 that represents the percentage of features correctly implemented and tested. A factor of 1 correspond to a solution where all the functional features required by the MMP are correctly implemented/tested.