



# Università degli Studi di Messina

Progetto Programmazione Web e Mobile

# Introduzione e analisi degli strumenti utilizzati

**Smart Parking** è una Web App che permette agli utenti di acquistare biglietti per le soste a pagamento nella città di Reggio Calabria.

Il progetto è sviluppato principalmente in **PHP**, utilizzando il Framework **Laravel**, che garantisce una struttura robusta e flessibile per il codice.

I vari parcheggi disponibili sono mostrati graficamente tramite la mappa fornita da **OpenStreetMap**, resa interattiva grazie all'utilizzo della libreria **Leaflet**.

L'utente ha la possibilità di effettuare login e registrazione tramite gli appositi form, per registrarsi é necessario fornire:

1. *Nome*
2. *Cognome*
3. *Email*
4. *Password*

I parcheggi della città di Reggio Calabria sono divisi per “zona”, ogni zona é caratterizzata dalla sua tariffa:

1. ***Zona A, 2 €/ora***
2. ***Zona B, 1 €/ora***
3. ***Zona C, 0.5 €/ora***

La zona A rappresenta la parte più “centrale” della città, la zona C la parte più “decentrata” mentre la zona A una via di mezzo.

I parcheggi sono rappresentati nella mappa da “cerchietti” (disegnati tramite Leaflet) e possono assumere due stati:

***1. Blu, parcheggio libero***

***2. Rosso, parcheggio occupato***

Ogni parcheggio é identificato da un numero di lotto che lo identifica e può essere occupato solamente da una macchina la volta.

L'utente per effettuare la sosta deve necessariamente registrare un veicolo indicando:

***1. Nome del veicolo***

***2. Targa del veicolo***

Questo e' necessario sia per i controllori, sia per mostrare nella mappa da chi il parcheggio e' realmente occupato.

Per acquistare un parcheggio l'utente dovrà avere credito sufficiente nel proprio account, per aggiungere del credito basta recarsi nella pagina dedicata dove andrà semplicemente inserito l'importo da aggiungere.

Fatto ciò per acquistare un parcheggio basterá selezionarne uno dalla mappa, si verra' indirizzati nella pagina di acquisto dove andrà indicato:

***1. Inizio della sosta (inserito automaticamente dal sistema)***

***2. Fine della sosta***

***3. Veicolo col il quale si parcheggia***

Una volta acquistato il parcheggio in base la tariffa, lo stato del parcheggio si aggiornerà nella mappa colorandosi di rosso e verrà sottratto dal credito il costo totale di esso.

Il costo del parcheggio é dato dalla seguente formula:

$$\text{costo} = \text{tempo}_i * \text{prezzo}_i$$

Es: 2h di sosta in Zona C (0.5 €/ora) equivalgono a un costo di 1€.

Il sistema ogni **10s** verificherá (tramite un **cron job**) i parcheggi scaduti, che verranno resi disponibili nuovamente, se un parcheggio risulta occupato l'interfaccia non ne permette l'acquisto.

L'utente tramite la pagina relativa allo stato del proprio parcheggio puó terminare in anticipo la sosta, tramite il tasto “**end parking**” (questo non costituisce però un rimborso dei minuti non utilizzati).

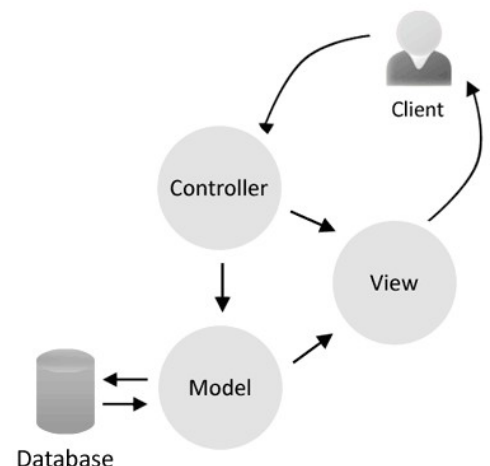
## Perché Laravel?

Laravel è un framework PHP moderno che offre numerose funzionalità per semplificare e ottimizzare lo sviluppo di applicazioni web. Tra le sue principali caratteristiche, troviamo:

### 1. Architettura MVC (Model-View-Controller)

Laravel utilizza l'architettura **MVC**, un pattern che divide il codice in tre componenti principali:

- **Model:** gestisce i dati e la loro logica di business.
- **View:** si occupa della presentazione grafica dei dati.



- **Controller:** gestisce la logica applicativa e coordina le interazioni tra Model e View.

## 2. Routing semplice e flessibile

Laravel offre un sistema di routing intuitivo che permette di definire facilmente le rotte delle applicazioni, sia per pagine statiche che dinamiche.

## 3. ORM Eloquent

L'ORM (Object-Relational Mapping) **Eloquent** permette di interagire con il database in modo semplice e diretto, utilizzando modelli che rappresentano le tabelle del database, riducendo la necessità di scrivere complesse query SQL.

## 4. Sistema di autenticazione e autorizzazione

Laravel include un sistema integrato per la gestione di utenti, autenticazione e autorizzazione, semplificando la protezione delle risorse e la gestione dei permessi.

## 5. Blade Templating Engine

**Blade** è il motore di template di Laravel, che consente di scrivere codice HTML dinamico in modo elegante e performante, supportando funzionalità come l'ereditarietà dei layout.

## 6. Artisan Console

**Artisan** è la console di comando di Laravel, che permette di eseguire operazioni come:

- Creazione di modelli, controller e migrazioni.
- Gestione del database.
- **Automazione di task personalizzate.**

## 7. Supporto per API RESTful

Laravel semplifica la creazione di API RESTful grazie a strumenti come i controller di risorse e la gestione delle rotte API.

### Interfaccia Client

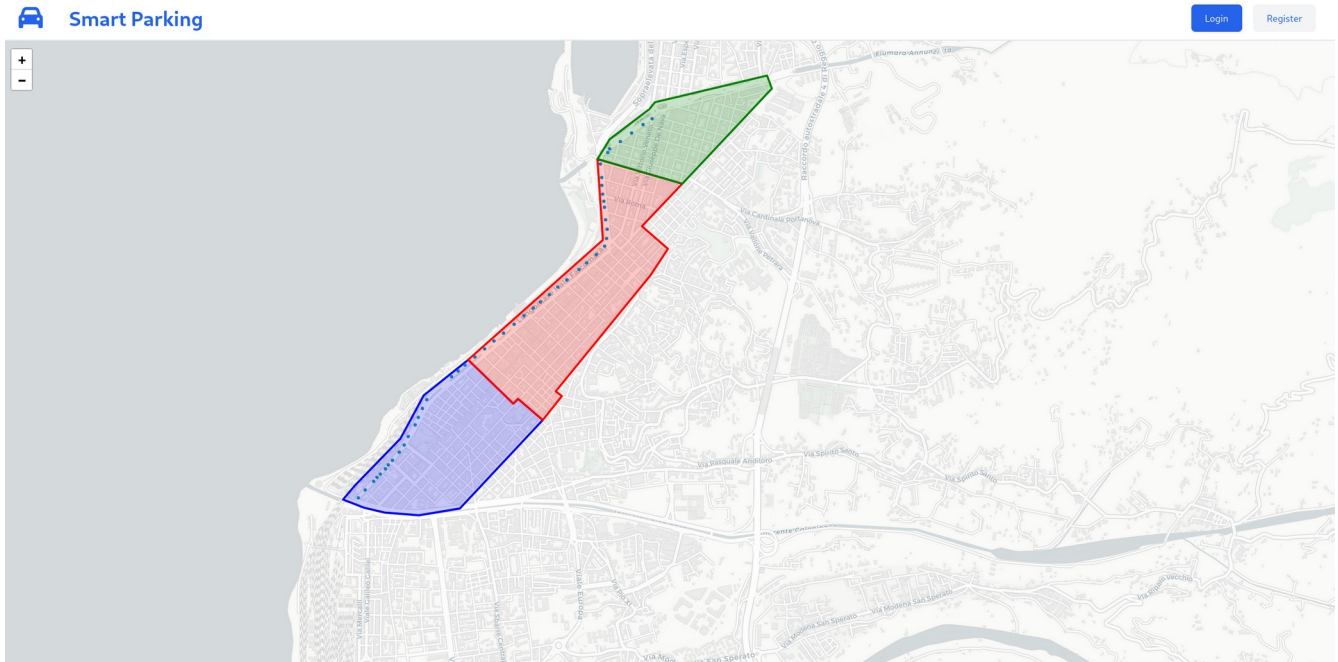


Figure 1: Homepage

### Start parking today!

Access your account to top up your balance, purchase parking tickets, and view real-time availability with our interactive map. Convenient, fast, and secure experience smarter parking today.



No account? [Register here](#)

[Log in](#)

Figure 2: From Login


### Welcome to Smart Parking!

Create your account to access convenient parking solutions in Reggio Calabria. Register now to top up your balance, purchase tickets, and enjoy real-time updates on parking availability through our interactive map. Parking made simple and hassle-free!

By creating an account, you agree to our [terms and conditions](#) and [privacy policy](#).

[Create an account](#)[Already have an account? Log in](#)

Figure 3: Form Registrazione

 Smart Parking

Welcome back, Davide [Logout](#)

Map  
Profile ▾

## Your profile

Edit your personal information below

First Name

davide

Surname

ferrara

Email

davide98ferrara@gmail.com

Password

Confirm edit



 Davide  
davide98ferrara@gmail.com

Figure 4: Pagina profilo

 Smart Parking

Welcome back, Davide [Logout](#)

Map  
Profile ▾

Credit Page

Your current credit: 0.00€

-

1

+

Buy credit


 Davide  
davide98ferrara@gmail.com

Figure 5: Pagina credito



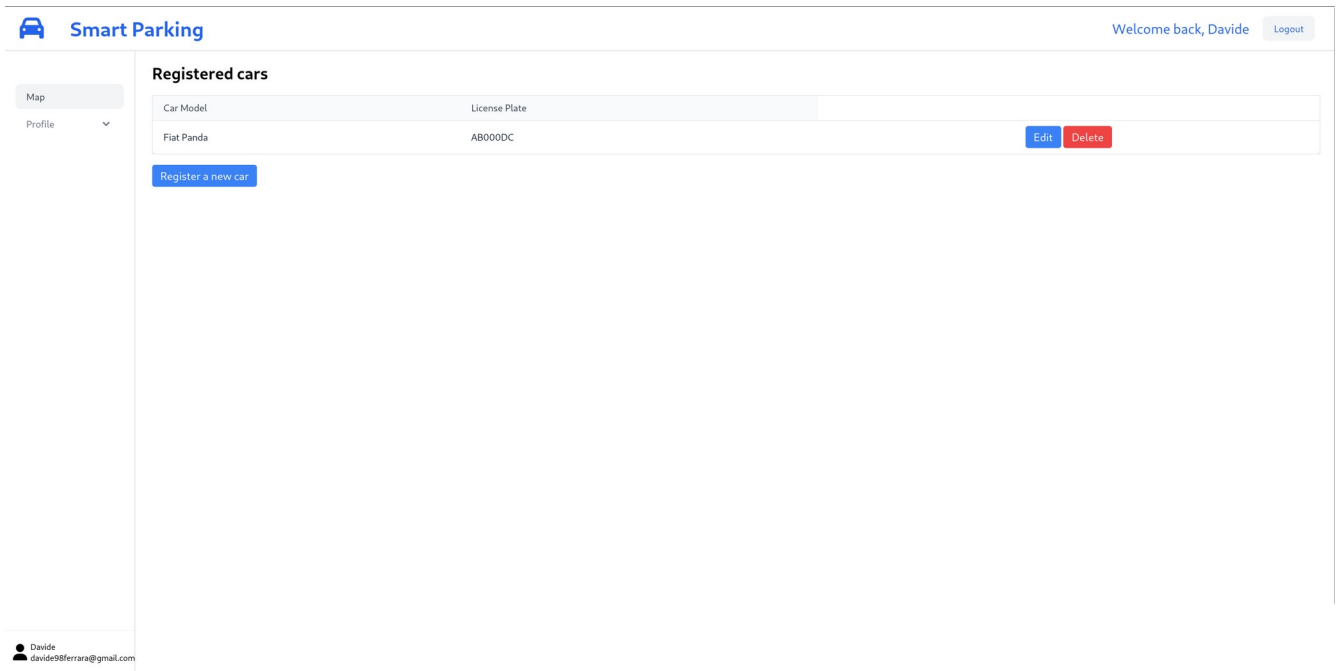


Figure 6: Pagina Veicoli utente

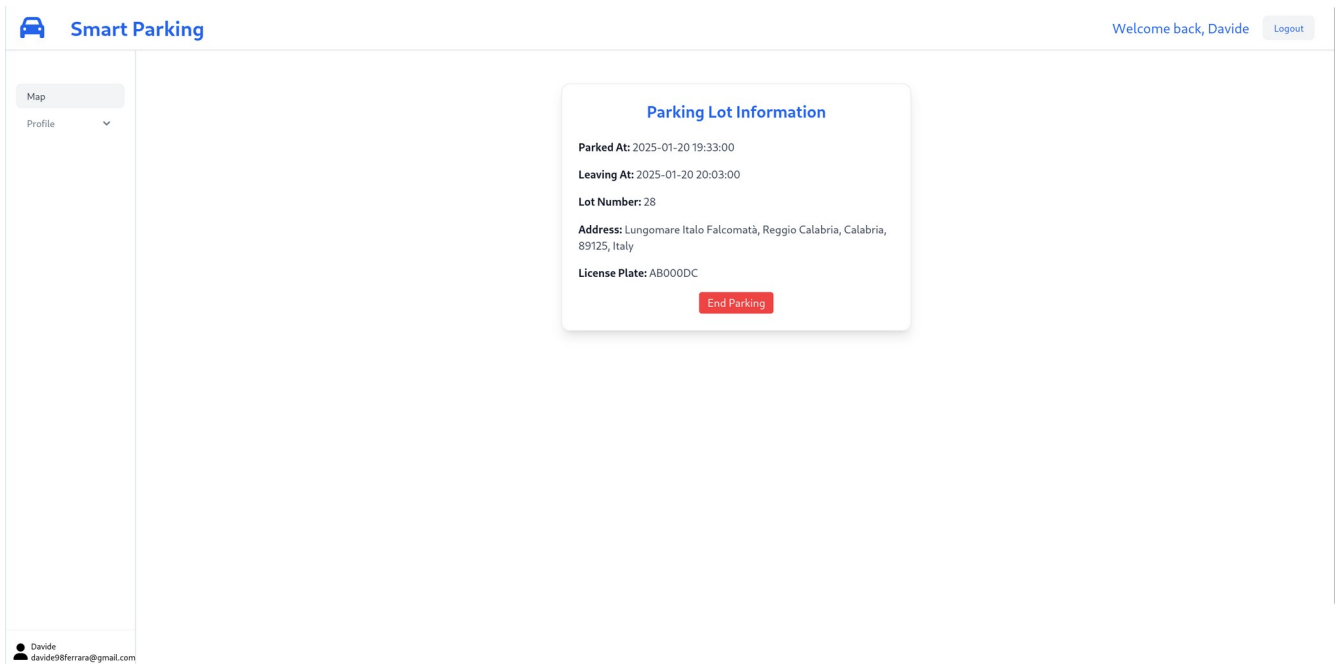


Figure 7: Informazioni parcheggio

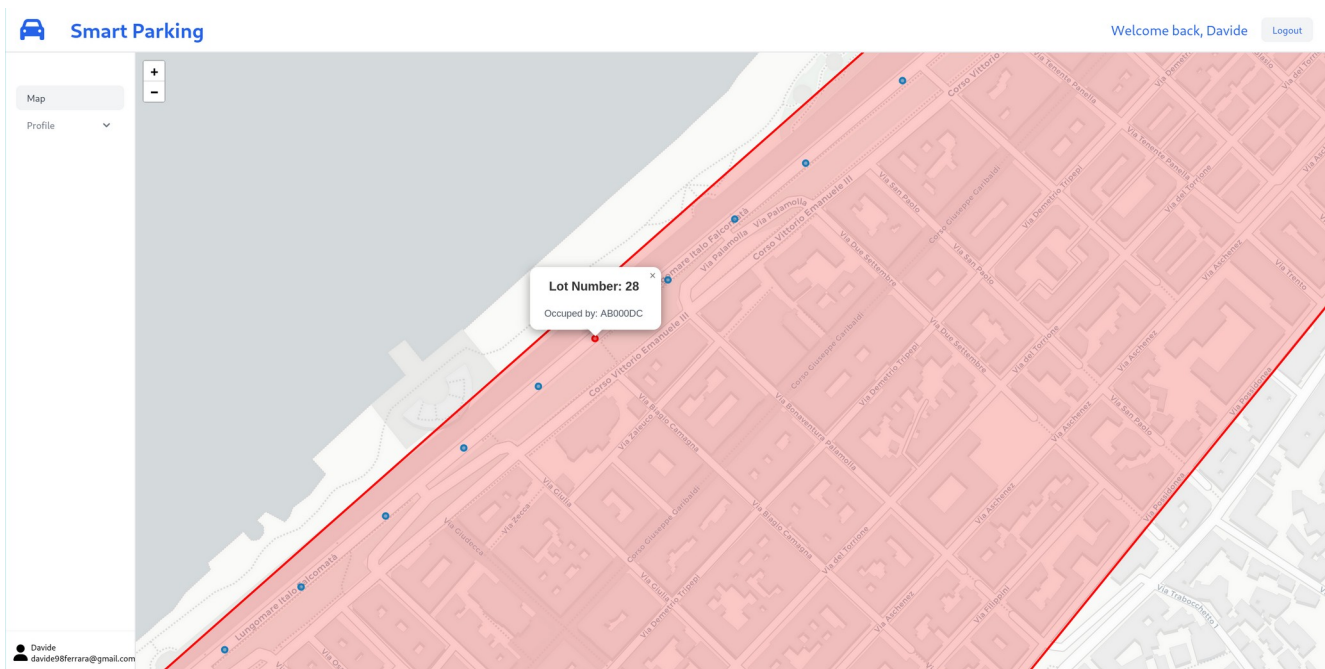


Figure 8: Stato del parcheggio

## Pannello di controllo amministratore

In Smart Parking troviamo anche la possibilità di poter creare, editare e rimuovere eventuali zone di sosta tramite il pannello di controllo accessibile solo da un amministratore.

Tramite il pulsante “**pick from map**” e’ possibile selezionare un punto nella mappa che andrà a completare i restanti campi automaticamente.

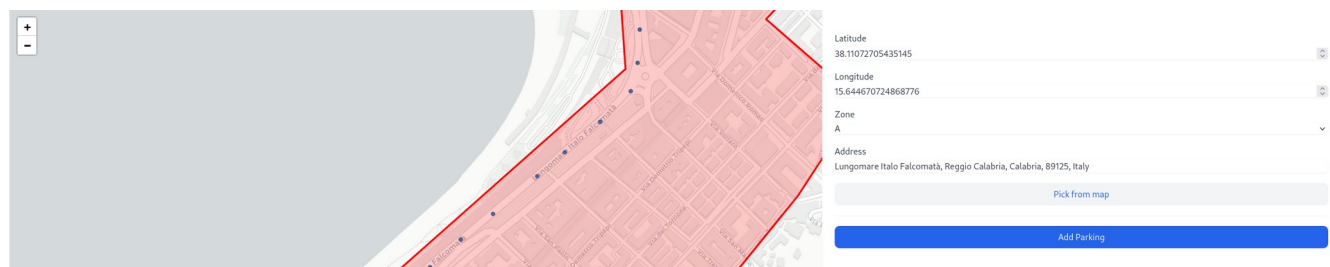


Figure 9: pannello admin per aggiungere un parcheggio al database

Parking Lots						
Lat	Long	Lot Number	Status	Zone ID	Address	Actions
38.107925489115686	15.640490534739094	18	0	2	Lungomare Italo Falcomatà, Reggio Calabria, Calabria, 89125, Italy	<a href="#">Edit</a> <a href="#">Delete</a>
38.108208410529290	15.640898261803004	19	0	2	Lungomare Italo Falcomatà, Reggio Calabria, Calabria, 89125, Italy	<a href="#">Edit</a> <a href="#">Delete</a>

Figure 10: pannello admin per editare ed eliminare parcheggi dal database

## Modello Relazionale

Table 1: User Table

id	BIGINT (unsigned)	Primary Key
name	VARCHAR	NOT NULL
surname	VARCHAR	NOT NULL
phone_number	VARCHAR	UNIQUE, NULLABLE
email	VARCHAR	UNIQUE, NOT NULL
email_verified_at	TIMESTAMP	NULLABLE
password	VARCHAR	NOT NULL
remember_token	STRING	NULLABLE
created_at	TIMESTAMP	NULLABLE
updated_at	TIMESTAMP	

Table 2: Session Table

id	STRING	Primary Key
user_id	BIGINT (unsigned)	Foreign Key (nullable)
ip_address	VARCHAR (45)	NULLABLE
user_agent	TEXT	NULLABLE
payload	LONGTEXT	NOT NULL
last_activity	INTEGER	NOT NULL

**Table 3: Parking Lot Table**

Colonna	Tipo di Dato	Proprietà
lot_number	INT (unsigned, auto-increment)	Primary Key
lat	DECIMAL (precision, scale)	NOT NULL
lng	DECIMAL (precision, scale)	NOT NULL
curr_status	BOOLEAN	DEFAULT false
occupied_by	BIGINT (unsigned, nullable)	Foreign Key (nullable) che fa riferimento a <code>users.id</code> , onDelete: set null
license_plate	VARCHAR (7)	NULLABLE
zone_id	BIGINT (unsigned)	DEFAULT 1, Foreign Key che fa riferimento a <code>parking_lot_zones.id</code> , onDelete: cascade
address	STRING	NULLABLE
created_at	TIMESTAMP	NULLABLE
updated_at	TIMESTAMP	NULLABLE

**Table 4: Parking Lot Zones Table**

Colonna	Tipo di Dato	Proprietà
id	BIGINT (unsigned, auto-increment)	Primary Key
letter	CHAR (1)	NOT NULL
price_per_hours	DECIMAL (4, 2)	NOT NULL
created_at	TIMESTAMP	NULLABLE
updated_at	TIMESTAMP	NULLABLE

**Table 5: Parking Lot Histories Table**

Colonna	Tipo di Dato	Proprietà
id	BIGINT (unsigned, auto-increment)	Primary Key
user_id	BIGINT (unsigned)	Foreign Key che fa riferimento a <code>users.id</code> , onDelete: cascade
lot_number	INT (unsigned)	Foreign Key che fa riferimento a <code>parking_lots.lot_number</code> , onDelete: cascade
start_parking	DATETIME	NOT NULL
end_parking	DATETIME	NOT NULL
processed	BOOLEAN	DEFAULT false
processed_at	DATETIME	NULLABLE
created_at	TIMESTAMP	NULLABLE
updated_at	TIMESTAMP	NULLABLE

**Table 6: Car Table**

Colonna	Tipo di Dato	Proprietà
id	BIGINT (unsigned, auto-increment)	Primary Key
model_name	VARCHAR	NOT NULL
license_plate	VARCHAR (7)	NOT NULL, UNIQUE
created_at	TIMESTAMP	NULLABLE
updated_at	TIMESTAMP	NULLABLE

**Table 7: User Credit**

Colonna	Tipo di Dato	Proprietà
user_id	BIGINT (unsigned)	Primary Key, Foreign Key che fa riferimento a <code>users.id</code> , onDelete: cascade
total	DECIMAL (6, 2)	DEFAULT 0
created_at	TIMESTAMP	NULLABLE
updated_at	TIMESTAMP	NULLABLE

## HTML 5

Il progetto utilizza le linee guida di HTML 5 quindi l'utilizzo di **tag semantici** come *nav*, *main*, *aside*, *content*, *article*, ecc..

Sotto possiamo visionare il file “**layout,blade.php**” che é la componente comune di tutte le pagine **HTML** grazie al templating offerto da Blade.

```
<!DOCTYPE html>
<html>

<head>
    @if (request()->is('/'))
        <title>Smart Parking</title>
    @endif

    @if (request()->is('register/*'))...@endif
    @if (request()->is('login/*'))...@endif
    @if (request()->is('admin/*'))...@endif
    @if (request()->is('profile/*'))...@endif
    @if (request()->is('parking/*'))...@endif
    @if (request()->is('admin/parking/add') || request()->is('/'))...@endif

    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swap" rel="stylesheet">
    <link rel="stylesheet" href="{{ asset('css/app.css') }}">
    <link rel="stylesheet" href="https://site-assets.fontawesome.com/releases/v6.7.2/css/all.css">

    @vite('resources/css/app.css')
</head>
```

Figure 11: *layout.blade.php*

```

<body>
    <!-- Navbar -->
    <header class="bg-white">
        <nav class="mx-auto max-w px-4 sm:px-6 lg:px-8 border-b-2">
    </header>

    <div class="flex">
        @auth
            <aside class="h-screen flex flex-col border-r-2 min-w-48 justify-between border-e bg-white">
        @endauth

        <main class="w-full h-screen bg-white">
            {{ $slot }}
        </main>
    </div>

    <footer class="bg-white">
</body>
</html>

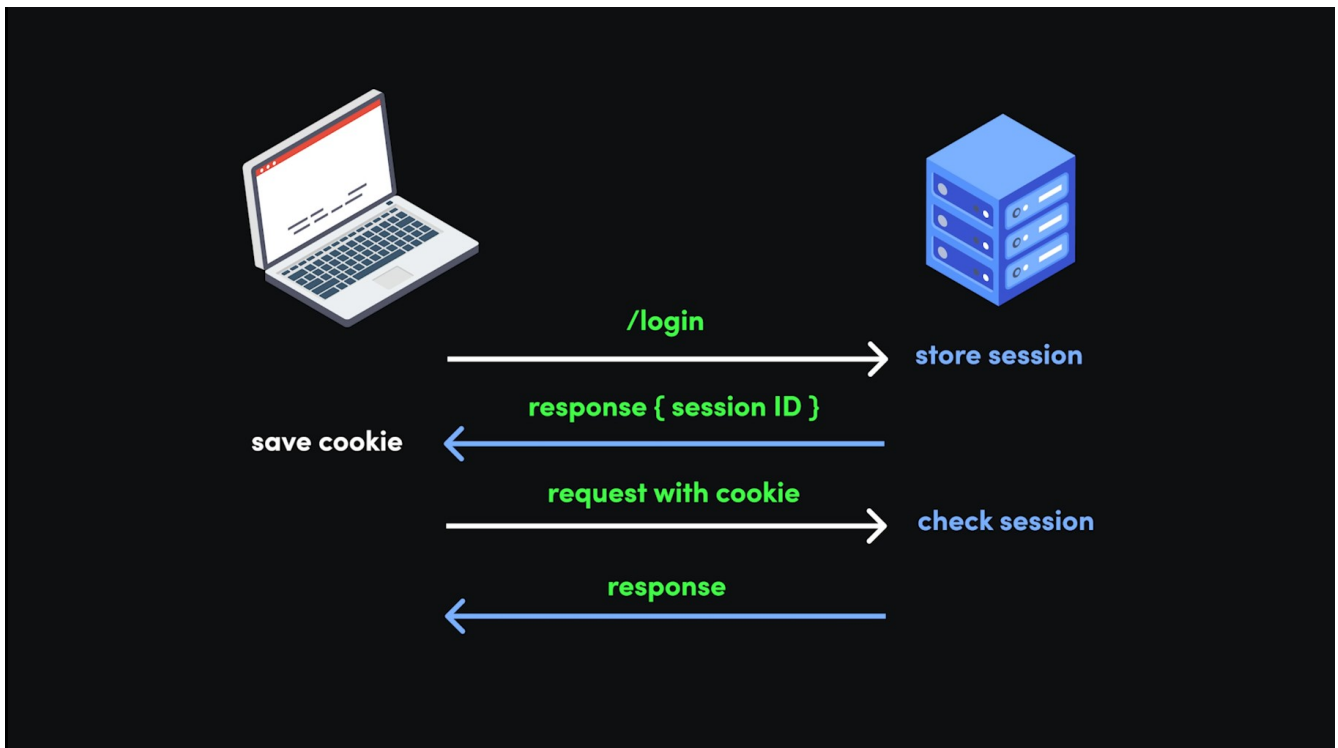
```

Figure 12: *layout.blade.php*

## Autenticazione e sicurezza

Il metodo predefinito di Laravel per la gestione dell'autenticazione é basato su **sessioni**:

**Login:** Quando un utente effettua il login con successo, Laravel crea una sessione sul server, memorizza l'ID dell'utente autenticato e successivamente (il server) invia un cookie di sessione al client.



**Cookie di sessione:** Questo cookie contiene un identificatore di sessione (ad esempio, `smart_parking_session`) che il browser del client memorizza. Ogni volta che il client effettua una richiesta al server, il cookie viene inviato automaticamente, consentendo al server di identificare l'utente autenticato.

```
▼ Data
▼ smart_parkings_session:"eyJpdil6lmtDRnlrRL...liwidGFnljoiIn0%3D"
  Created:"Tue, 21 Jan 2025 09:56:09 GMT"
  Domain:"localhost"
  Expires / Max-Age:"Tue, 21 Jan 2025 11:56:41 GMT"
  HostOnly:true
  HttpOnly:true
  Last Accessed:"Tue, 21 Jan 2025 09:56:41 GMT"
  Path:"/"
  SameSite:"Lax"
  Secure:false
  Size:364
```



Laravel di default fornisce anche un sistema di sicurezza contro attacchi **CSRF (Cross-Site Request Forgery)** per le richieste che modificano lo stato dell'applicazione (come login e registrazione).

Ogni modulo deve includere un **token CSRF** che Laravel verifica automaticamente, specificato all'interno del codice HTML tramite il tag **@csrf**.

```
<form action="/login" method="POST" class="mx-auto mb-0 mt-8 max-w-md space-y-4">
@csrf
  <div>
    <label for="email" class="sr-only">Email</label>

    <div class="relative">
      <input
        name="email"
        type="email"
        class="w-full rounded-lg border-gray-200 p-4 pe-12 text-sm shadow-sm "
        placeholder="Enter email"
      />
    </div>
  </div>
</form>
```

Figure 13: Token CSRF

## API

L'insieme di rotte sono definite rispettivamente in due file: **routes/web.php** e **routes/api.php**.

In web.php vengono definite rotte specifiche per l'interfaccia web e sono assegnate al **web middleware group**, che fornisce come visto primo sessioni e protezione CSRF.

```
// Auth
Route::get(uri: '/login', [SessionController::class, 'create']);
Route::post(uri: '/login', [SessionController::class, 'store']);

Route::post(uri: '/logout', [SessionController::class, 'destroy'])->middleware(middleware: 'auth');

Route::get(uri: '/register', [RegistredUserController::class, 'create']);
Route::post(uri: '/register', [RegistredUserController::class, 'store']);
```

Figure 14: web.php

In api.php possiamo invece specificare endpoint stateless.

```
Route::get( uri: '/parking-lots', function () {
    Log::info( message: "[api.php] Sending parking lots json");
    return ParkingLot::all();
});
```

Figure 15: api.php

```
JSON Raw
▶ 0: Object { lot_number: 18, lat: "38.107925489115686", lng: "15.640490534739094", ... }
▶ 1: Object { lot_number: 19, lat: "38.108208410529290", lng: "15.640898261803004", ... }
```

Figure 16: parking\_lot.json

## Pianificazione delle task (crontab job)

Per avviare la web app basta eseguire da riga di comando lo script bash “**start.sh**”, esso conterrà il necessario per avviare i **container docker** e il **cron job**.

```
#!/bin/bash

./vendor/bin/sail up -d

docker compose exec laravel.test bash -c "service cron start"

docker compose exec laravel.test bash -c "crontab /var/www/html/cronfile"

docker compose exec laravel.test bash -c "npm run dev --host"

exit
```

Figure 17: start.sh

Nei sistemi operativi Unix e Unix-like, il comando **cron** consente la pianificazione di comandi, ovvero la registrazione di questi presso il sistema per essere poi mandati in esecuzione periodicamente in maniera automatica dal sistema stesso.

```
root@4ab19cc71f5c:/var/www/html# cat cronfile
* * * * * cd /var/www/html && ./update_parking_status.sh >> /dev/null 2>&1
root@4ab19cc71f5c:/var/www/html#
```

Figure 18: cronfile

Tramite il seguente cron job possiamo eseguire lo script “**update\_parking\_status.sh**” ogni minuto, lo script non fa altro che eseguire il comando artisan sei volte, una volta terminato verrà rimpiazzato da un altro cron job che rieseguirà lo stesso script, facendo in modo che la tabella del database relativa allo stato dei parcheggi si aggiorni quindi ogni **10s**.

```
#!/bin/bash

for i in {1..6}; do
    php /var/www/html/artisan updateParkingStatusJob
    sleep 10
done
```

Figure 19: update\_parking\_status.sh

```

function checkParkingLotStatus($parkingLotHistory) {

    $now = Carbon::now()->setTimezone( timeZone: 'Europe/Rome');
    $endParking = $parkingLotHistory->end_parking;
    $lotNumber = $parkingLotHistory->lot_number;

    Log::info( message: "Checking: now at: " . $now . " end at:" . $endParking . " lot: " . $lotNumber);

    // Se il parcheggio risulta scaduto lo rendo nuovamente disponibile
    if ($now > $endParking) {
        DB::beginTransaction();
        try {
            DB::table( table: 'parking_lots')->where( column: 'lot_number', $lotNumber)->update([
                "curr_status" => 0, "occupied_by" => null, "license_plate" => null
            ]);

            DB::table( table: 'parking_lot_histories')->where( column: 'id', $parkingLotHistory->id)->update([
                'processed' => true,
                'processed_at' => $now,
            ]);

            DB::commit();
        } catch (\Exception $e) {
            DB::rollBack();
            Log::error( message: "Error updating parking lot status: " . $e->getMessage());
        }
    }
}

```

Figure 20: Funzione per controllare lo stato di un parcheggio

```

Artisan::command( signature: 'updateParkingStatusJob', function () {

    $parkingLotHistories = DB::table( table: 'parking_lot_histories')->where( column: 'processed', operator: false)->get();

    foreach ($parkingLotHistories as $parkingLotHistory) {
        checkParkingLotStatus($parkingLotHistory);
    }

})->purpose( description: 'Update the parking status inside the DB');

Schedule::command( command: 'updateParkingStatusJob');

```

Figure 21: Artisan comand

## Aggiornamento lato Client

Dopo aver aggiornato i dati presenti nel nostro database basandoci sull'orario di inizio e fine della sosta in un parcheggio dobbiamo fare in modo che l'utente possa visualizzare ciò senza dover aggiornare manualmente la pagina, vengono in nostro aiuto quindi le **richieste asincrone (AJAX)** e le **Promise**.

Le Promise sono utilizzate in JavaScript per gestire operazioni asincrone, nel codice sotto mostrato viene creato un nuovo oggetto **XMLHttpRequest**, che è utilizzato per effettuare richieste HTTP.

La richiesta viene inviata all'endpoint **“/api/parking-lots”** tramite il metodo **send()**, successivamente viene definita la funzione di callback che andrà a risolvere o rigettare la promise.

```
Show usages  davide-ferrara
getParkingLots() : Promise<unknown> {
  return new Promise( executor: (resolve, reject) : void => {
    var xhttp : XMLHttpRequest = new XMLHttpRequest();
    xhttp.open( method: "GET", url: "/api/parking-lots", async: true);
    xhttp.send();

    xhttp.onreadystatechange = function () : void {
      if (this.readyState === 4) {
        if (this.status === 200) {
          resolve(JSON.parse(this.responseText));
        } else {
          reject(new Error("Request failed with status: " + this.status));
        }
      }
    };
  });
}
```

Figure 22: Promise

La promise viene utilizzata all'interno della funzione **“updateParkingLots”**, se la promise é risolta allora gli elementi che rappresentano i parcheggi nella UI vengono aggiornati senza che la pagina venga aggiornata manualmente dall'utente.

```

Show usages  👤 davide-ferrara +1
const updateParkingLots = () : void => {
  console.log('[Parking Map] Starting Updating...');

  if (this.parkingsList.length > 0) this.removeAllParkings();

  this.getParkingLots() Promise<unknown>
    .then(parkingLots => { ... }) Promise<void>
    .catch(error => {
      console.log(error);
    })
  console.log('[Parking Map] Parking lots updated!');
};

updateParkingLots();
setInterval(updateParkingLots, this.refreshRate);
}

```

Figure 23: funzione per aggiornamento visivo client

In fine la funzione di aggiornamento tramite “**setInterval**” viene schedulata per essere eseguita ogni 10s, seguendo quindi il refresh rate del backend.

## Window: setInterval() method



**Baseline** Widely available



The `setInterval()` method of the `Window` interface repeatedly calls a function or executes a code snippet, with a fixed time delay between each call.

Figure 24: setInterval MDN docs