Toni’s Garage — API Documentation

Class Team Project - December 2025

12/11/2025

Table of Contents

1. Introduction

The Toni’s Garage API provides the backend services for the Toni’s Garage car marketplace platform. It enables frontend applications to retrieve vehicle listings, view detailed car information, manage user accounts, handle featured content, and record payment attempts. All functionality is delivered through a single entry file (api.php) using an action-based routing system.

Responses are returned in structured JSON, making the API easy to work with across different environments such as browser-based frontends, mobile apps, or testing tools like Postman. Most endpoints require an API key for security, and strict same-origin CORS rules ensure that only trusted clients can access protected resources.

This documentation explains how each endpoint works, the required parameters, and the expected responses. It serves as a practical reference for developers integrating the Toni’s Garage backend into their applications.

2. Overview

The Toni’s Garage API provides all core data operations needed for the marketplace application. Through its action-based request system, clients can retrieve structured information about vehicles, users, and payments with a consistent JSON response format. The API supports the following major capabilities:

- **Retrieve a full listing of cars** for use in search results, inventory pages, or catalog displays.
- **Retrieve detailed information about a specific vehicle**, including images, features, seller details, engine specifications, and transmission data.
- **Retrieve vehicle types**, allowing clients to populate filters, categories, or navigation menus.
- **Retrieve featured vehicles**, supporting promotional sections or highlight displays within the frontend UI.
- **Create payment records** for purchases, deposits, or transaction tracking.
- **Register new users** with basic profile information.
- **Authenticate returning users**, enabling login sessions and personalized frontend interactions.

All responses are returned as JSON objects containing either a `data` payload or a structured error message. Most endpoints require an API key supplied in the `X-API-Key` header, ensuring controlled access to sensitive vehicle and user data. The only exception is the payments endpoint, which intentionally remains open to support direct form submissions.

The API is designed to be lightweight and predictable, making it suitable for a variety of client interfaces while maintaining clear separation between backend operations and frontend presentation.

3. Security & Authentication

3.1. API Key

Most endpoints require **X-API-Key**. This key acts as the primary method for controlling access to protected resources such as vehicle data, user accounts, and internal marketplace information. When a request is received, the API retrieves the supplied key from the request headers and validates it using:

hash_equals(API_KEY, trim($key));

The use of hash_equals() provides a timing-safe comparison, reducing the risk of timing attacks. If the API key is missing, malformed, or does not match the expected value, the request is immediately rejected. In such cases, the API responds with a JSON error message:

{ "ok": false, "error": "Unauthorized", "debug_received_key": "..." }

3.2. CORS

To protect the API from unauthorized cross-origin requests, the server enforces a strict same-origin policy. When generating the response headers, the API dynamically sets:

Access-Control-Allow-Origin: https://<current-host>

This means:

- Only web pages served from the **same domain and protocol** as the API may issue requests.
- Third-party websites, external tools, or applications running on different origins cannot access the protected endpoints unless CORS rules are manually expanded.
- This behavior adds an additional layer of security by ensuring that only trusted frontends can interact with the API, even if the API key were somehow exposed.

These restrictions are especially important for browser-based clients, helping to prevent CSRF-style attacks and unauthorized data access.

4. Base URL

The Toni’s Garage API is built around a single entry point, api.php, which processes all incoming requests. Rather than exposing multiple route files, the API uses an action query parameter to determine which internal operation should be executed. This keeps the system lightweight and easy to deploy across different environments.

The base URL for accessing the API varies depending on where the project is hosted.

4.1. Production (Solace server)

When deployed to the RIT Solace server, the API is available at:

https://solace.ist.rit.edu/Toni-s_Garage_Final/backend/api.php

4.2. Local development

During development or testing on your own machine, the API is accessed through:

http://localhost/Toni-s_Garage_Final/backend/api.php

In this mode, the API defaults to local database credentials, allowing developers to test functionality without affecting production data.

5. General Request Pattern

The Toni’s Garage API follows a simple and predictable request structure. Every operation is identified by an action query parameter, which tells the API which internal process to execute. Depending on the endpoint, additional query parameters, JSON payloads, or form fields may also be required.

This design centralizes routing within a single file (api.php), reducing complexity and ensuring consistent handling of authentication, headers, and responses across all endpoints.

5.1. Request Types

The API supports two primary HTTP methods:

5.1.1. GET Requests

Used for retrieving data. These requests typically require only the action parameter and, when necessary, additional query parameters such as id.

GET requests are used for:

- Listing all vehicles
- Retrieving detailed information for a specific vehicle
- Retrieving vehicle categories
- Viewing featured vehicles

Example GET request:

```
/api.php?action=vehicle&id=3
```

5.1.2. POST Requests

Used for sending data to the server. POST requests typically include JSON payloads or form fields depending on the endpoint requirements.

POST requests are used for:

- User signup
- User login
- Submitting payment information

Example POST scenario:

```
POST /api.php?action=signup
Content-Type: application/json
```

POST endpoints validate request bodies and return detailed error messages when required fields are missing.

6. Endpoints

6.1 Get Vehicle Details

The **Get Vehicle Details** endpoint returns comprehensive information about a specific vehicle in the Toni’s Garage inventory. This includes all major attributes stored in the database, allowing frontends to present complete product pages with rich details such as specifications, seller information, and images.

This endpoint performs multiple joins across related tables (images, types, users, engine, transmission, interior) to produce a fully assembled vehicle object in a single response.

6.1.1 Data Included

This endpoint returns:

- **Basic information:** name, model, description, documentation (doc), ownership status
- **Pricing data:** price, number of vehicles sold
- **Color options:** up to four colors
- **Images:** primary vehicle image and interior image
- **Vehicle type:** ID and type name
- **Seller information:** seller name and email
- **Features:** fuel type, mileage
- **Engine details:** engine name, type, horsepower
- **Transmission:** name and type

This makes it the most detailed endpoint in the API.

6.1.2. Headers

The endpoint requires a valid API key:

X-API-Key: YOUR_SUPER_SECRET_KEY_HERE

Without this header, the API will return a 401 Unauthorized response. And this goes for all the following requests.

6.1.3. Query Parameters

The id parameter must be a positive integer. Passing 0, non-numeric values, or omitting the parameter triggers a 400 error.

Name	**Required**	**Description**
action	yes	Vehicle
id	yes	Vehicle ID (integer)

6.1.4. Example Request

This request retrieves the full details for the vehicle with ID **1**.

```
curl -X GET \
  -H "X-API-Key: YOUR_SUPER_SECRET_KEY_HERE" \
  "https://your_host/api.php?action=vehicle&id=1"
```

6.1.5. Success Response

Below is the actual structure returned by the API when a vehicle is found:

```
{
 "ok": true,
 "data": {
   "vehicle_id": 1,
   "name": "Toyota Camry",
   "model": "2022",
   "doc": "...",
   "price": 25000,
   "owned_before": false,
   "description": "A great sedan.",
   "number_sold": 12,
   "color_1": "Red",
   "color_2": "",
   "color_3": "",
   "color_4": "",
   "vehicle_image_url": "https://...",
   "type_id": 3,
   "type_name": "Sedan",
   "user_id": 7,
   "seller_name": "Jane Doe",
   "seller_email": "jane@example.com",
   "feature_id": 21,
   "fuel": "Gasoline",
   "mileage": 12345,
   "engine_id": 4,
   "engine_name": "V6",
   "engine_type": "Petrol",
   "horse_power": 203,
   "transmission_id": 2,
   "transmission_name": "Automatic",
   "transmission_type": "6-speed",
   "interior_image_url": "https://..."
 }
}
```

- This is a **fully assembled data object** created by joining multiple tables.
- Fields such as engine_type, transmission_type, and interior_image_url are included only because the code performs inner/outer joins on matching tables.
- owned_before indicates whether the car is pre-owned.
- number_sold is useful for analytics or “popular vehicles” features.

6.1.5. Errors

Status	**Response**
400	{ "ok": false, "error": "Missing or invalid ID" }
401	{ "ok": false, "error": "Unauthorized" }
404	{ "ok": false, "error": "Vehicle not found" }

- Invalid or missing id values never reach the database query—they are rejected immediately.
- Unauthorized requests always return 401, regardless of other parameters.
- If the vehicle ID passes validation but no record exists, the API returns 404.

6.2 List All Vehicles

The **List All Vehicles** endpoint returns a summarized list of all vehicles stored in the Toni’s Garage database. This endpoint is optimized for page listings, search results, category browsing, or grid displays where only essential information is required.

It does **not** return deep relational data like engine specifications or seller details. Instead, it provides high-level attributes suitable for preview cards or catalog layouts.

6.2.1. Data Included

Each vehicle entry includes:

- Basic identifying information: name, model, unique ID
- Pricing information
- Whether the car was previously owned
- Vehicle type name
- Primary image URL
- Fuel and mileage from the features table

6.2.2 Example Request

```
curl -X GET \
  -H "X-API-Key: YOUR_SUPER_SECRET_KEY_HERE" \
  "https://your_host/api.php?action=vehicles"
```

6.2.3 Success Response

```
[
 {
   "vehicle_id": 1,
   "name": "Toyota Camry",
   "model": "2022",
   "price": 25000,
   "doc": "...",
   "owned_before": false,
   "type_name": "Sedan",
   "vehicle_image_url": "https://...",
   "mileage": 12345,
   "fuel": "Gasoline"
 }
]
```

Notes

- The results are ordered by vehicle_id in ascending order.
- All images and type names are retrieved through JOINs but kept minimal for listing efficiency.
- This endpoint is ideal for building product lists and search screens.

6.3 List Vehicle Types

The **List Vehicle Types** endpoint returns all available vehicle categories along with a count of how many vehicles belong to each category. This information is typically used to populate filters, dropdown menus, or category sections within the frontend.

The count is generated using a LEFT JOIN, ensuring that categories with zero associated vehicles still appear in the results.

6.3.1 Example Response

```
[
 {
   "type_id": 1,
   "type_name": "SUV",
   "vehicles_available": 5
 },
 {
   "type_id": 2,
   "type_name": "Truck",
   "vehicles_available": 2
 }
]
```

- vehicles_available is a dynamic count from the database.
- Results are ordered alphabetically by type name.
- Useful for UI filtering, sidebar navigation, and analytics.

6.4 List Featured Vehicles

The **List Featured Vehicles** endpoint returns vehicles that have been specially flagged as featured using the is_featured field in the database. These vehicles typically appear on homepage carousels, promotional banners, or spotlight sections.

This endpoint returns a curated subset of each vehicle’s data to keep responses compact while still providing meaningful detail.

6.4.1. Data Included

Each featured vehicle includes:

- Basic identifying fields
- Price and description
- Main image URL
- Number sold
- A features JSON object with:

- air conditioning (ac)
- model year
- transmission type
- smart screen availability

These fields are selected to highlight the selling points of each vehicle.

6.4.2. Example Response

```
[
  {
    "id": 5,
    "name": "Tesla Model X",
    "price": 90000,
    "description": "Electric luxury SUV.",
    "image_url": "https://...",
    "number_sold": 3,
    "features": {
      "ac": true,
      "model": "2023",
      "transmission": "automatic",
      "smart_screen": true
    }
  }
]
```

- features is returned as a MySQL JSON_OBJECT, bundled into a single field.
- Ideal for homepage highlights or carousel displays.
- This endpoint intentionally contains fewer fields than /vehicle to optimize frontend load times.

6.5 Submit Payment

The **Submit Payment** endpoint records a basic payment attempt in the database. This is typically used for checkout flows, reservation submissions, or deposit confirmations.

Unlike other endpoints, this route is intentionally left **unauthenticated** so that public-facing payment forms can submit data without requiring an API key.

6.5.1. Form Fields

Field	**Required**	**Description**
credit_holder_fname	yes	Cardholder first name
credit_holder_lname	yes	Cardholder last name
last_four	yes	Last four digits of the card

- The endpoint automatically cleans the card number input by removing non-digit characters and then extracts only the last four digits.

6.5.2. Behavior on Success

If all required fields are present and the database insert succeeds, the API:

1. Creates a new record in the payments table
2. Issues an HTTP redirect to: ./frontend/confirmation.php

Because of the redirect, the API does **not** return a JSON success response.
Instead, the user is sent directly to the confirmation page.

This behavior is intentionally designed for form-based submissions where a redirect is part of the workflow

6.5.3. Errors

If one or more required fields are missing:

```
{ "ok": false, "error": "Missing payment fields" }
```

If a database or server error occurs:

```
{ "ok": false, "error": "Payment insert failed", "debug": "Detailed error message" }
```

6.6 User Login

The User Login endpoint authenticates a user by validating their email and password. If the credentials match a record in the database, the API returns the user's profile information.

6.6.1. Request Body

```
{
  "email": "jane@example.com",
  "password": "secret"
}
```

Both email and password fields are required. Credentials are checked **as plain text** against the database.

6.6.2. Success Response

```
{
  "ok": true,
  "data": {
    "user_id": 7,
    "role_id": 3,
    "first_name": "Jane",
    "last_name": "Doe",
    "email": "jane@example.com",
    "phone_number": "555-1234",
    "address": "123 Main St",
    "state": "NY",
    "zip_code": "14623",
    "country": "USA"
  }
}
```

The password is **never returned** in the response.

6.6.4 Errors

Missing fields – 400

```
{ "ok": false, "error": "POST method required" }
{ "ok": false, "error": "Email and password are required." }
```

Invalid credentials – 401

```
{ "ok": false, "error": "Invalid email or password." }
```

Wrong HTTP method – 405

6.7 User Signup

The **User Signup** endpoint registers a new user by inserting their basic information into the database. The API accepts a full name and automatically splits it into first and last names.

6.7.1. Request Body

```
{
  "full_name": "Jane Doe",
  "email": "jane@example.com",
  "password": "secret",
  "confirm_password": "secret"
}
```

6.7.2. Field Behavior

- full_name is split into first and last components
- password and confirm_password must match exactly
- Email must not already exist in the database

6.7.3. Success Response

```
{
  "ok": true,
  "data": {
    "user_id": 12,
    "first_name": "Jane",
    "last_name": "Doe",
    "email": "jane@example.com"
  }
}
```

This response confirms that the account was successfully created.

6.7.4 Errors

Missing fields – 400

```
{ "ok": false, "error": "POST method required" }
{ "ok": false, "error": "All fields are required." }
```

Password mismatch – 400

```
{ "ok": false, "error": "Passwords do not match." }
```

Email already exists – 409

```
{ "ok": false, "error": "Email already registered." }
```

Wrong method – 405

7. Error Handling

The Toni’s Garage API is designed to provide clear, consistent, and predictable error responses so that client applications can handle failures gracefully. Regardless of the endpoint, all errors follow the same JSON structure and use standard HTTP status codes. This ensures that both human developers and automated systems can understand and react to error conditions reliably.

7.1.1 Standard Error Response

Whenever an error occurs—whether due to invalid input, missing authentication, or internal server problems—the API responds with a JSON object structured like this:

```
{

  "ok": false,

  "error": "Description of the error"

}
```

- **ok is always false** for error states.
- **error contains a human-readable explanation**, making debugging easier.
- Additional fields (such as debug) may appear during development, but should be removed in production environments.

This consistency allows client applications to automatically detect failures simply by checking the ok flag.

7.1.2. HTTP Status Codes

In addition to the JSON structure, the API uses standard HTTP status codes to communicate the category of the error. Clients can rely on these codes for conditional logic, error displays, and automated flows.

Status Code	**Meaning**
400 - Bad Request	Missing or invalid parameters
401 - Unauthorized	Missing or incorrect API key
404 - Not Found	Requested resource does not exist
405 - Method Not Allowed	Wrong HTTP method used
409 - Conflict	Data conflict such as duplicate error during sign up
500 - Server Error	Database issues or unexpected failures

7.1.3. Error Predictability

Every endpoint follows this unified structure, ensuring that frontend clients can:

- Detect errors reliably
- Display accurate messages to users
- Implement consistent error-handling logic

8. Security Considerations

While the Toni’s Garage API includes basic security features, there are several important considerations for safe deployment and future improvement.

8.1 API Key Handling

API keys protect most endpoints, but the current implementation returns the received key in error responses for debugging. This should be removed in production, as it exposes sensitive information.

8.2. Password Storage

User passwords are stored and compared in plain text. This is not secure and should be replaced with industry-standard hashing (e.g., password_hash() and password_verify()).

8.3. CORS Restrictions

The API only allows requests from the same domain. This prevents unauthorized cross-site requests but also means:

- External frontends cannot access the API
- You must update CORS rules if deploying a separate frontend

8.4. Input Validation

Form and JSON inputs are minimally validated. Additional filtering and sanitation are recommended to prevent malformed data or injection attacks.

8.5. Payment Endpoint Exposure

The payments endpoint intentionally requires no authentication, but this means:

- It can be accessed by anyone who knows the URL
- Rate limiting, CAPTCHA, or token-based validation may be necessary in production environments

8.6. Database Credentials

The API switches between production and local credentials based on hostname. Ensure that:

- Credentials are not exposed in public repositories
- Environment variables or secure configuration files are used in the future

9. Summary

The Toni's Garage API provides a centralized, consistent, and extensible interface for all backend operations within the car marketplace system. Built using PHP and MySQL, the API serves as the communication layer between frontend applications and the underlying database. It delivers structured JSON responses, employs predictable error handling, and operates on a simple action-based routing pattern, making it easy to integrate into a variety of client environments.

Through its set of endpoints, the API enables consumers to retrieve vehicle listings, query detailed information for individual vehicles, manage user authentication and registration, obtain metadata such as vehicle types, and submit payment information. The system enforces API-key authentication on all endpoints except payments, thereby protecting sensitive user and vehicle data while still allowing flexibility for publicly triggered payment submissions. The API is tightly coupled with CORS restrictions to ensure requests originate only from trusted domains, improving security and reducing risks of unauthorized access.

Each endpoint has been carefully documented in this reference to include its purpose, required parameters, expected responses, and error formats. The API's response structure adheres to a uniform JSON schema (`ok`, `data`, `error`), ensuring consistency across all operations. This makes it easier for frontend developers to predict behavior and handle success/failure states effectively.

While the API is functional and suitable for controlled environments such as academic or internal use, developers should be aware of current limitations and areas designated for improvement if the system is intended for production deployment. These include password hashing, removal of debugging information, hardening of the authentication mechanism, improved validation, and proper handling of sensitive payment details. Implementing these enhancements will significantly increase the security and reliability of the service.

In conclusion, the Toni's Garage API forms the backbone of the whole project, enabling seamless interaction between the user interface and persistent data. This documentation provides developers with the clarity and precision they need to build, test, extend, and maintain applications that rely on the API.