Javascript Technologies
TUM
Date: 12 February 2023
Seminar Project

Mercedes x Salesforce Javascript Seminar

Team Backend

Dan Bachar [1], Bryan Smith Collazos Duran [1], Ihsan Soydemir [1], Hasan Huseyin Kacmaz [1]

[1] School of Computation, Information and Technology, Technical University of Munich, Munich, 85748, Germany

Abstract

Motivation: Build a backend serving the Mercedes & Salesforce Car Configurator for the Javascript Seminar
Results: Deployment of backend was successful
Availability: Backend is available under https://mercedes-backend.herokuapps.com/api
Contact: dan.bachar@tum.de
Supplementary information: Supplementary data are available online on the *GitHub* repository.

1 Intro

This paper summarizes our efforts in creating a scalable, future proof, and robust backend service that supports and answers requests from the Mercedes Car Configurator frontend, realised in a RESTful manner. We discuss how, using the Salesforce tool Heroku, we are able to run both client and server in parallel on the same machine. Illustrated next are the different technologies we've used, the project's architecture and its different modules. Following that we'll walk through current limitations and future prospects for the project. This project has the potential to change how Mercedes interacts with its customers, primarily through the use of Artificial Intelligence and natural language models.

2 Architecture

We've used a multi-tier architecture to design our code. Client requests are first handled by controllers, that first perform validations on the requests to make sure it's well formed. When requests pass our different validations they are then passed on to the next layer, the business logic. The business logic layer consists of services who process the requests, sometimes already returning a response. Some requests that need to access the database are forwarded to the data access layer, which is mostly represented by repositories that access the database. See figure 1 for an architecture diagram that presents our project.

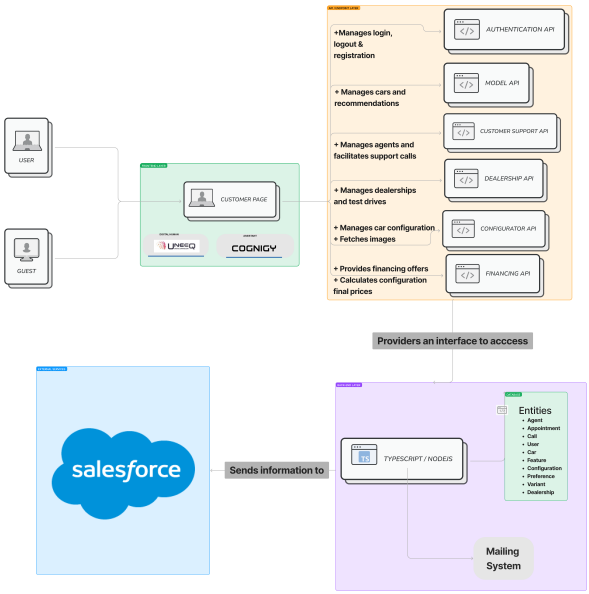

Fig. 1. Architecture Diagram

3 Technologies

A combination of cutting-edge technologies was used in the creation of the system described in this paper to deliver a reliable and scalable solution. The project used Docker, a top platform for containerizing applications, to make sure the system could be easily deployed and maintained. Utilizing Nest.js, a progressive Node.js framework that offers a solid development environment for creating scalable server-side applications, the back-end logic was implemented.

The project used PostgreSQL and Redis, two essential technologies, to manage and store the data. The system's structured data was stored using PostgreSQL, a potent open-source relational database management system. The system's cache was managed by Redis, an in-memory data structure store, which also offered quick access to frequently used data.

By utilizing these technologies, the development team was able to deliver a solution that could be deployed quickly and easily. It also ensures that the system is scalable and efficient. The combination of Docker, Nest.js, PostgreSQL and Redis ensures a robust and scalable technological foundation for the development of the system described in this paper.

3.1 API

An API (Application Programming Interface) was used in the development of the system described in this paper to facilitate communication between its components. It was designed to provide a simple and standardized way for the system to interact with web application. The API was implemented using REST (Representational State Transfer) architecture, a widely adopted standard for building APIs. REST offers a clear and flexible approach for creating APIs, making it easy to consume and understand the API's functionality.

The API allowed for easy integration with external systems and provided a simple and consistent interface for accessing the system's functionality. External systems could easily interact with the system and retrieve required data without having to understand the underlying implementation details. The API played a crucial role in the development of the system, enabling the development of robust and flexible integrations. In addition, it made easier the exchange of data among different components.

3.2 Rest API

A RESTful API is a type of API that uses HTTP requests fired from the client to fetch and interact with resources on the server. This allows for the client and server to be hosted on different machines, although this is not a decision we've made, as performance was prioritised higher than modularity. REST stands for Representational State Transfer, which refers to a set of constraints that are applied to the design of the API, such as a client-server layered architecture usually utilizing HTTP stateless requests and cacheable responses. RESTful APIs allow for communication between systems on the web and enable the retrieval of data or the execution of actions through HTTP requests using common methods such as GET, POST, PUT, and DELETE, which are very similar to traditional CRUD operations on a data storage. RESTful APIs are stateless, meaning that each request contains all the information necessary to complete the request, and their response is self-contained, providing all the information necessary to understand the response. These reasons and more have contributed to the popularity of Rest APIs, making them the de-facto most used technology in developing APIs.

3.3 Javascript

A high-level, dynamic, and interpreted programming language is JavaScript. The creation of desktop and mobile applications as well as server-side scripting are all common uses for JavaScript, one of the most well-liked programming languages in the world.

In this paper, JavaScript was utilized to deliver a comprehensive and engaging user experience on the front-end and also to provide server-side functionality. On the front-end side, JavaScript was used to handle user inputs. Javascript also perform real-time data validation and trigger animations and transitions.

On the back-end, JavaScript was leveraged through the use of Nest.js. Nest.js provides developers with the ability to run JavaScript on the server-side, making it a popular choice for building scalable, high-performance web applications and it also supports Typescript. In this paper, Nest.js was used to develop the back-end logic, handle API requests, and integrate with the PostgreSQL database.

The versatility of JavaScript is one of its key advantages, as it can run on multiple platforms and can be utilized for a wide range of applications. The language's popularity has resulted in the creation of a vast ecosystem of libraries and frameworks, which has greatly reduced the time and effort required to develop applications. This is particularly beneficial in a research context, as developers can leverage existing solutions to address common problems and focus their efforts on the unique aspects of their projects.

3.4 Typescript

An open-source, statically typed programming language called TypeScript extends JavaScript with new functionality to make development easier. To overcome JavaScript's drawbacks and give programmers a more organized and maintainable codebase, TypeScript was created.

In this study, TypeScript was used to speed up the development process and guarantee the codebase's scalability and maintainability. The statically typed nature of TypeScript makes it simpler for developers to understand and maintain the code because it enables the early detection of type-related errors and offers better documentation and code navigation. In addition, TypeScript offers features like classes, interfaces, and type inference that JavaScript lacks but which are essential for creating complex applications.

3.5 Nodejs

JavaScript code may be executed outside of a web browser using Node.js, an open-source, cross-platform runtime environment. It is based on the V8 JavaScript engine and provides an event-driven, non-blocking I/O architecture, making it a well-liked option for developing scalable, high-performance server-side applications. Additionally, Node.js has a sizable and vibrant developer community that adds to the vast ecosystem of modules that can be quickly deployed and utilized to enhance applications. It has become a popular technology for a variety of applications, including web servers, chatbots, and microservices, thanks to its adaptability and simplicity.

We decided to utilize Node.js for the Mercedes project because it gave us a quick and easy way to create a real-time online application. We were able to manage many concurrent connections thanks to its event-driven architecture, and its robust ecosystem of modules gave us the resources we needed to swiftly create and launch our application. Additionally, because Node.js employs JavaScript as its primary programming language, we were able to apply the expertise we already had, which sped up and streamlined the development process.

3.6 Nestjs

Nest.js is a potent framework for Node.js developers to use to create scalable and maintainable server-side applications. It provides a modular, dependency-injection based design that makes it simple to organize and test code. It is built on top of the well-known Express.js web framework. Nest.js is a flexible option for a variety of use scenarios since it supports a variety of other widely used libraries and technologies, such as TypeScript, MongoDB, and WebSockets. With features like automated module discovery, built-in support for testing, and dependency injection that assist decrease boilerplate code and increase code maintainability, its design philosophy places a high priority on developer productivity and code quality.

Because of its strong support for Object-Relational Mapping (ORM) libraries, we decided to choose Nest.js as our server-side framework. We wanted a framework that would let us operate with our PostgreSQL database fast and simply, and Nest.js's support for TypeORM made it the perfect option. We were able to write clear, maintainable database code without having to worry about low-level details like SQL queries and connection management thanks to TypeORM's powerful features, which include its support for transaction management and ability to generate

database schema based on TypeScript classes. This made Nest.js the ideal fit for the needs of our project, along with its simplicity of use and modular architecture.

3.7 Docker

Docker is a platform for containerizing applications that was utilized in the development of the system described in this paper. Docker provides a standardized and isolated environment for applications, which makes it easy to deploy and maintain the system. The development team was able to ensure that the system was consistently deployed and configured across different environments thanks to utilization of Docker. It also minimizes the risk of compatibility issues. Docker's containerization approach also provides a high level of security, as containers are isolated from each other and from the host system. This isolation reduces the risk of security breaches and makes it easier to manage the system's security.

Docker was a key technology utilized in the development of the system described in this paper. Its ability to provide a consistent and isolated environment for applications made it an essential component in the delivery of a robust and secure solution.

3.8 Postgres

PostgreSQL is an open-source relational database management system known for its reliability, data integrity, and advanced features. It can be used for a wide range of applications, from small projects to large-scale, high-traffic web applications. PostgreSQL has a large and active community of developers, users, and contributors, who help to ensure its continued improvement and evolution over time. The reason why we chose Postgres was it's ease of use and setup, full docker support and the fact that it's open sourced, and thus can be used fror Mercedes

3.9 Swagger

The development of the backend application described in this paper uses the API testing and documentation tool called by Swagger. Developers may easily comprehend the functionality of the API and test the API's endpoints with the help of Swagger, which offers a clear and interactive description of the API.

The development team was able to test the API's endpoints to make sure they were functioning properly and rapidly and simply write documentation for the API using Swagger. This minimized the possibility of defects or compatibility problems while also considerably simplifying the process of creating and maintaining the API.

Additionally, Swagger offered customers an easy-to-use interface for virtualizing the API's endpoints, enabling the development team to swiftly make changes and overview the API's functionality and design. This made it simpler to check the functionality of the API and find any areas that needed improvement.

3.10 Redis

Redis is an in-memory database that is lightning fast and designed for real-time data processing. It supports a variety of data structures such as strings, objects, lists, sets, and more, making it ideal for use cases such as caching and frequent requests whose responses don't often change. Redis' ease of use and performance make it a popular choice for developers who need a fast and flexible data store for their applications. We've used Redis to store the user authentication tokens which were stored as JSON Web Tokens, because we correctly anticipated that the most common API call would be to check if the user is authenticated. Saving the tokens on the database would be roughly 1000 times slower, as portrayed in the diagram. To support our claim of choosing Redis over Postgres for authentication, we've ran a benchmark test that pitches the former against the latter, simulating an increasing amount of parallel connections (1-50) on the x-axis, and measuring the achieved number of transactions per second on the y-axis.

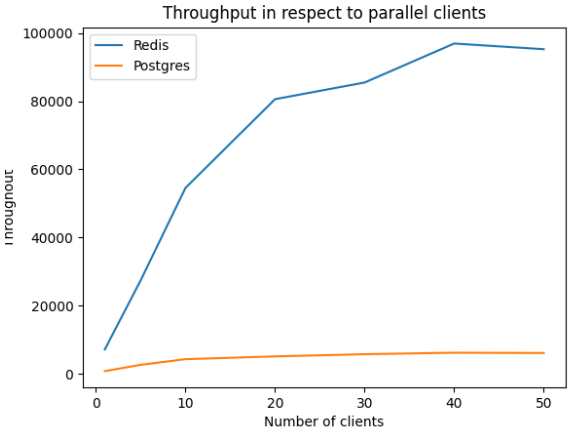

Fig. 2. Throughput of Redis in comparison to Postgres

4 Current limitations

Choosing the current architecture as the implemented project architecture does have its price. The project is used as a presentation of concept for Mercedes, and won't be able to support hundreds of parallel connections, for example.

4.1 Docker

By using Docker over a bare-metal machine we've made a conscious decision to sacrifice performance in favor of ease-of-use, uniform environment across different development computers and the production machine on Heroku. A bare-metal computer hosting the server code would have been faster, due to the abstractizations posed by Docker.

4.2 Heroku

Hosting the server on a cloud provider such as Heroku is extremely easy, and involves almost no configuration work. However a different provider could have been chosen which favors performance and load balancing over ease of use. Heroku also charges by usage, and scales really badly when the number of uses increase. A different cloud provider, or self hosting, would make this reasonably cheaper.

5 Modules

In this section we are going to cover one critical module used in the project.

5.1 Images

5.1.1 Problem statement

One of the main problems the project had, was figuring out a way in which we could get car objects from the front-end and then efficiently retrieve the images of the cars hosted in back-end. The solution had to take into account different aspects, chief among them:

1. The diversity among different models.

2. The ease for uploading the images to the database.
3. The performance to recover the images from the database.
4. The flexibility to create car images in different views. e.g. Interior, exterior, or other features added in long-term.

5.1.2 The structure string

In order to solve P.S. 1 and after multiple approaches, the most suitable one was working with structured entities that would allow us to pick the image up from the database.

Listing 1. a simple car example

```
{
 "car": {
  "model": "Model EQS"
  ...
 },
 "exteriorConfiguration": {
  "features": [
   {
    "featureType": "RIM",
    "value": "LIGHT-ALLOY WHEELS IN 5 TRIPLE SPOKE",
    ...
   }
  ]
 },
 "interiorConfiguration": {
  "features": [
   {
    "featureType": "UPHOLSTERY",
    "value": "NAPPA LEATHER BLACK-SPACE GRAY",
    ...
   }
  ]
 },
 ...
}
```

As every car (see example 1) will have the same structure, we can define a common hierarchy to get the images, in which relevant car parameters (in this case, "featureType" or "model") will determine the returned image. We are going to call the final string *"STRUCTURE STRING"*.

e.g. if for the exterior images we define:

```
structure: string = model/.../RIM/...
```

Once the structure string is parsed by the values, we get:

```
parsedStructure: string = Model EQS/.../LIGHT-ALLOY WHEELS IN 5 TRIPLE SPOKE/...
```

Because the parsed structure string can refer to multiple images, with the same car configuration, we can consider this string as a unique ID for a set of car images.

5.1.3 The File Explorer as DB

To address requirements 2 and 3, we found a solution that could associate a set of images with our structure string. Given that our project only involves a single car, it made little sense to set up an entire database just for this purpose. Instead, we discovered that we could use the file explorer to categorize the images into sub-folders, and then create a static public folder to serve these images. This approach allows for quick and easy access to the images by any user.

http://my.server/MODEL EQS/.../LIGHT-ALLOY WHEELS IN 5 TRIPLE SPOKE/.../carimage.png

5.1.4 The Template Pattern

Finally, 4 requires our implemented system to support multiple views. To achieve this, we will create a template abstract class that defines the shared logic among all views. By using inheritance, each view can inherit this common logic and add its behavior as needed.

Listing 2. "Template class"

```
export abstract class ImgsCarTemplate {
  ...
  protected abstract readonly structureStringBase: string;
  ...

  //Establishes the target features from a VariantBody.
  protected abstract setFeatures(dto: bodyDTO): void;

  // Initializes object template.
  public async build(dto: bodyDTO) {
    ...
    this.setFeatures(dto);
    this.generateStructureString();
    ...
  }

  // Generates the parsed structure string from features.
  protected generateStructureString(): void {
    ...
    // Parse Structure String
    ...
  }


}
```

Listing 3. "Child class"

```
export class ImgsExtCar extends ImgsCarTemplate {
  protected readonly structureStringBase: string = 'COLOR/RIM/TYRE';
  ...

  protected setFeatures(dto: bodyDTO): void {
    this.features = dto.exteriorConfig.features;
  }
}
```

6 Methods

To manage development of the entire Mercedes project, to distribute and divide work amongst the different development teams, and to control the development process over time, we chose the Agile project management

methodology. Agile is a flexible, iterative approach to managing projects that values collaboration, customer satisfaction, and adapting to change. It emphasizes a flexible and responsive process, delivering working software incrementally and regularly reassessing and adjusting project goals and requirements using roadmaps. Agile methodologies, such as Scrum and Kanban, promote teamwork and continuous improvement, helping teams to deliver high-quality projects in a timely and efficient manner. Concretely we've used Scrum as our agile method of choice. We've held weekly sprint planning meetings, input sessions with the customers (Mercedes and Salesforce), and review meetings with the project leads.

7 Conclusion

In this paper we've presented our development process for Mercedes and Salesforce, alongside support from the project leads at the department for computer science and informatics at the TUM. We've explained major architectural decisions, technologies used, compared different databases for our use case and delved into the most important pieces of the code. We believe the project has been extremely helpful in learning how to use Javascript in the backend, gathering experience in project management and planning, working with customers and coordinating work over multiple teams.

Acknowledgements

- Amin Ben Saad
- Ronald Skorogobat
- Faris Ben Saad
- Dr. Guy Yachdav
- Prof. Dr. Burkhard Rost
- Prof. Dr. Hans-Joachim Bungartz