Technical design evaluation form

Evaluating team: **Champignons**

Original design authoring team: **Global Coders**

Recommended grade: **8**

Flaws and design defect

API endpoint naming and HTTP method usage does not follow industry standards

90% of endpoints were designed as POST, even though they should've been GET or PATCH requests. For example, to get orders the designed specified to POST /order/getList with body, even though this is just a read operation which does not mutate any resources. Such requests were changed to GET requests, excluding the verbs and exchanging body with route/query parameters.

Severity: Major

API is not versioned

Versioning the API allows existing users to continue using the older version without disruption while introducing new features or improvements in a separate version. A v1 prefix was added to all endpoints to tell the consumers that this is the first version of the API.

Severity: Minor

System does not have API layer

Presentation logic is coupled to business logic. There is no way for external entities to integrate with the system. API layer was introduced, which acted as a mediator between the frontend, written in React, and the business logic. This allows clients to be independent from the etechnology that the business logic was written in.

Severity: Major

Inconsistent naming in data models

Inconsistency brings chaos into the project. To solve it, all database columns are named using PascalCase. Therefore, programmers can focus on understanding the logic rather than deciphering names.

Severity: Minor

Money is stored as floats

Storing money as floats leads to rounding errors and precision issues. Float was changed to decimal to ensure correctness, compliance with financial standards, and avoid bugs that can arise from floating-point arithmetic.

Severity: Major

Product variants not implemented

Products have variation string list, which does not actually exist in the database ER diagram (inconsistency). The variations do not have price associated with them. Product modifiers were implemented to allow creating product variations.

Severity: Critical

Order items do not exist in the ER diagram

It’s impossible to know what items were ordered without storing them in the database. OrderItems table was added to the database.

Severity: Critical

Order item tax data is not stored

It’s impossible to know what tax with what rate was applied to the order item. Only the tax amount was stored. Order item tax was created, which persisted the tax name, rate and amount applied to the order for financial compliance.

Severity: Major

Service charges do not exist in the ER diagram

It is a requirement of the client to have service charges. They were added.

Severity: Critical

Discounts cannot be applied without pre-creating them

An employee is unable to apply a discount on the fly – it first must be created with a valid until date, which does not make sense when creating a discount for a single order. Discounts were split to predefined (created by the business owner) and flexible (created by the employee, when taking an order).

Severity: Major

Not working payment design

Payment design is bad. For example, to pay by gift card, the customer should provide gift card code and not the payment amount, as the API documentation specifies. To complete card payment, the backend needs to send client secret however it does not exist. Payments were split into 3 endpoints – cash, gift card and card – to make sure that one endpoint does only one thing.

Severity: Major

Missing split payment design

Split payment design is missing – there is no state to indicate that the order is partially and not fully paid. To fix it, order state Completed was introduced, meaning that order can no longer be changed and needs to be paid for. When order is fully paid for, it transitions to Closed state.

Severity: Critical

Credit card details are stored in the database

ER diagram suggests that clients credit card details should be stored in the database, which is a security flaw. To fix it, CC payments are processed by Stripe and the backend does not know anything about the CC details of the client.

Severity: Major

Passwords are stored in plain text

ER diagram suggests storing user passwords plain text, without hashing them. This is a security breach as all the user passwords would be exposed if the database got leaked. To prevent this, the passwords are stored hashed.

Severity: Critical.

Auto-increment IDs

Auto-increment IDs are a poor choice for large systems because they are not globally unique, leading to conflicts when merging data. They also expose predictable patterns, which can be a security risk when exposed externally.

Severity: Moderate

Service duration is stored as integer

Duration as an integer in the database is very ambiguous – it can be any time unit. It was changed to timespan.

Severity: Moderate

Document and YAML inconsistency

There are inconsistencies between PDF and YAML documentation. For example, service endpoints are non-existent in YAML documentation but exist in PDF.

Severity: Moderate

Incorrect data types in YAML

Properties like dateStart and dateEnd have the data type integer, they were changed to strings with date-time format, for better clarity and validation.

Severity: Minor

Incomplete flow diagrams logic

The documentation of flows like Reservation scheduling is incomplete as, according to the logic, after allowing modification, there is a checking if the reservation is valid. If it's invalid, an error is thrown, but if it's valid, the reservation is not saved but canceled.

Severity: Minor

Missing service endpoints

Swagger file does not contain endpoints for creating services, which can be used for reservations. We added them.

Severity: Major

Deviations from the original design

Data storage renamed to repositories

Data storage was renamed to repositories to follow industry standards. This helps new developers to onboard the project more quickly.

Changed IsPercentage field

IsPercentage field was used to indicate the discount type. However, it is unclear, since a discount can be of many types, e.g.: buy 1 get 1 free. Therefore, it was changed for an enum with values FixedAmount and Percentage.

Allowed discount to be applied to multiple products

According to the database ER diagram, a Discount can be applied only to a single product. Many-to-Many table was created to allow applying a single Discount to multiple products.

Added discount target

Added discount target field to indicate whether the discount should be applied to products or to the whole order.

Implemented optimistic locking for products and modifiers

Implemented optimistic locking for products and modifiers to ensure that inventory amount is modified correctly between concurrent requests.

Removed inventory endpoints

Inventory management is directly integrated in product/modifier endpoints.

Password storing

Passwords are stored hashed, not in plain text.

Unified user management

To simplify user management, there is one class ApplicationUser instead of SuperAdmin, Owner and Employee. These in database are mapped with the roles Admin, BusinessOwner and Employee accordingly.

Added reservation in progress state

Added reservation in progress state to indicate that it is currently being used by some order.

Used Tech-Stack

We chose to implement our backend system using ASP.NET instead of the suggested Spring Boot, and as for frontend, we used React instead of Angular.

Transition to SOA

We transitioned from the initial design, where entities such as Owner, Customer, and Employee contained methods for user creation, management, and authentication, to a service-oriented architecture. In our approach, entities like ApplicationUser are kept clean, serving as data holders, while business logic—such as creating users, managing roles, and handling logins/logouts—is delegated to dedicated service ApplicationUserService.

Authentication

Authentication endpoint v1/users/login accepts no riderictURI as we implmented OAuth2 Password flow without external authorization provider. This endpoint sets tokens as cookies that would not be available for JS and ensure additional security. If the credentials are incorrect, it returns HTTP 401 error code instead of 422, because it is not a validation error. Expiration time is set securely in JWT to avoid forging and is also not returned in the body. The body returns general user data – FirstName, LastName, Emails etc.

RefreshToken

V1/user/refresh accepts empty body and takes refresh token value from cookie.

Users Endpoints

V1/users endpoints contain all user-related management logic, including authentication and authorization, login, logout and update or delete. There is no separate getList endpoint that accepts POST requests, insted, v1/users accepts query parameters with pagination filter and optional businessId to filter data based on needs. Instead of POST for updating user data, the appropriate PATCH method is used.

Business Endpoints

No separate resources for each HTTP method, PATCH, POST, DELETE, GET are on the same v1/businesses resource with path parameter for business id if it is an individual entity action.

Delete Action on Business and ApplicationUser

Not to lose sensitive data, when user or business is deleted, user data in the database is anonymized to follow GDPR.

Customers Endpoint

The customer endpoint was not created as the system originally should not have customers as its users, so there is no need to have properties like password, and since customers' information is only needed if reservations are made their information like name, phone number were added to reservations.

Reservation information saving

Properties price and name were added to reservations in order to have full information of what service reservations were once issued even if service is already deleted.

Reservation information saving

Properties price and name were added to reservations in order to have full information of what service reservations were once issued even if service is already deleted.

Separated payments from orders API

Moved payments to separate endpoints to ensure that orders are not coupled to payments. Namely, creating a payment, adding a tip and refunding order payments.

Added more details to DTOs

Some DTOs did not contain details necessary for the frontend to fulfill the requirements. Therefore, we added some additional properties to DTOs.

Added background jobs

Background jobs were added to improve system resilience:

1. When an endpoint is called to refund an order, card order payments are marked as refund initiated. A background job runs periodically which completes the refunds. This means that API responses are faster, and refunds are more resilient to failure.
2. When a reservation is made, the message is marked as unsent. A background job runs periodically, which sends all messages that are unsent. Messages include idempotency key, which is sent to AWS SNS to ensure exactly once message delivery.
3. After the customer confirms the payment intent, frontend sends a request to the backend to check the payment status. However, if this request fails, then the backend does not know about the payment. To resolve this, a background job was added which periodically checks

payments, which are marked as pending in the database with Stripe. If a payment has been pending for more than 15 minutes, then it is cancelled.