BRIEF EXPLANATION

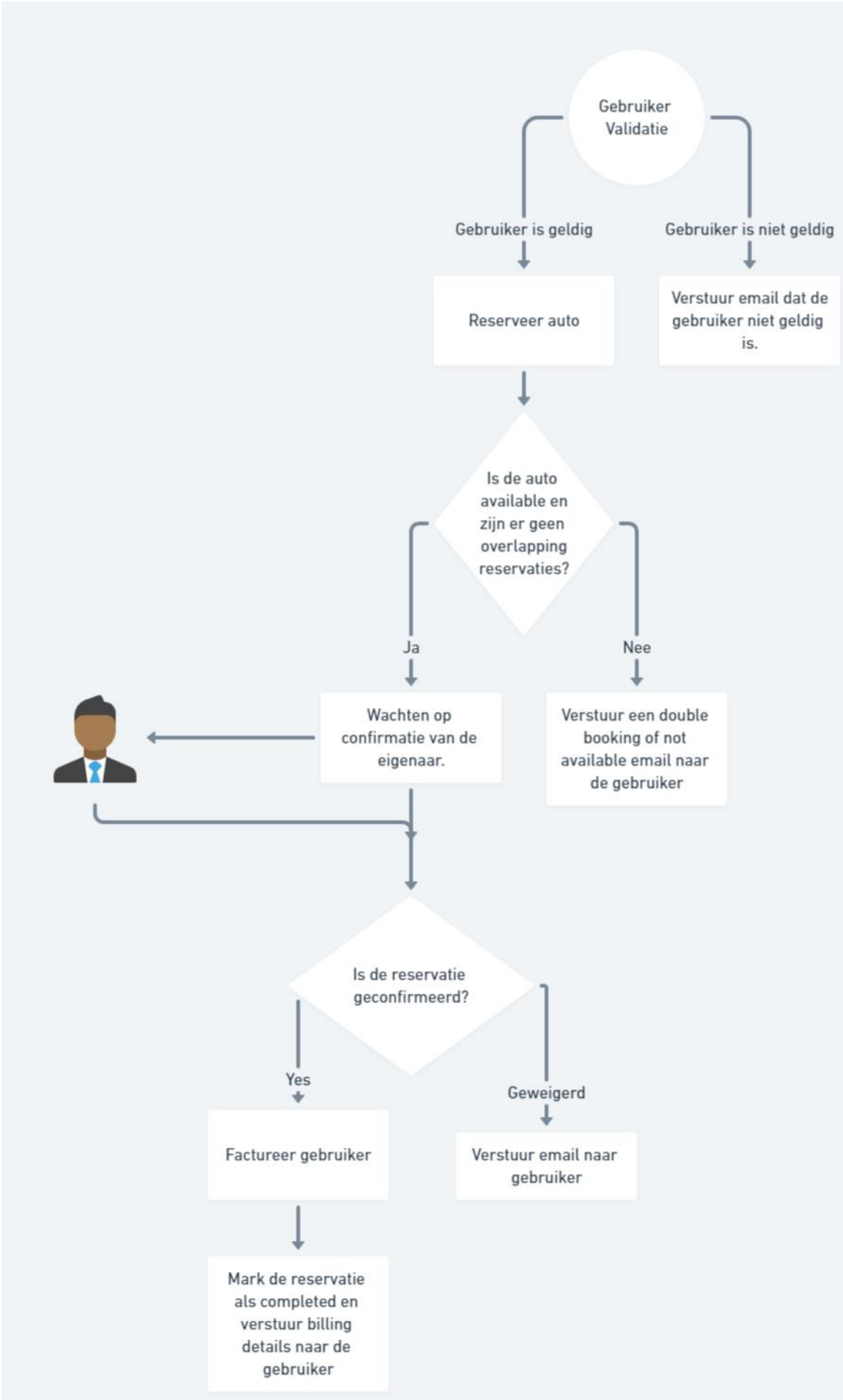
GEBRUIKTE PATTERNS:

- Service Discovery (Service Registry) Pattern
    - In ons project maken we gebruik van het Service Discovery (Service Registry) Pattern met een Eureka Server (@EnableEurekaServer) en Eureka Clients (@EnableDiscoveryClient). De Eureka Server is eigenlijk een centrale registratieplek, waarbij services zich aanmelden om hun beschikbaarheid kenbaar te maken. De Eureka Clients registreren zich bij de server en kunnen zo andere services ontdekken zonder iets te weten van de andere. Dit zorgt ervoor dat het systeem veel flexibeler en makkelijker uitbreidbaar is.
- Async Messaging (Publish/Subscribe) Pattern
    - We hebben de Async Messaging (Publish/Subscribe) Pattern met RabbitMQ geimplementeerd. We maken gebruik van queues en exchanges waar services naar luisteren. Wanneer een bericht (object) wordt ontvangen, voeren de services de vereiste taken uit en sturen ze een antwoord terug via het messaging-systeem. Hierdoor wordt asynchrone communicatie tussen verschillende delen van ons systeem mogelijk gemaakt.
- Saga Pattern (Orchestration)
    - In onze reservation service gebruiken we de Saga Pattern (orchestratie). We behandelen complexe transacties door berichten te sturen naar verschillende services, die vervolgens taken uitvoeren en reacties terugsturen. Een meer gedetailleerde uitleg over de specifieke werking zal later worden uitgelegd
- Dependency Injection Pattern
    - We passen het Dependency Injection Pattern toe in ons project, voornamelijk met behulp van "@Autowired". Deze annotatie wordt gebruikt om objecten intern te initialiseren.
- Circuit Breaker Pattern
    - In onze API-Gateway service implementeren we het Circuit Breaker Pattern met behulp van de CircuitBreakerFactory. Dit stelt ons in staat om alle inkomende verzoeken door te sturen naar de relevante services. Als er zich fouten voordoen, biedt dit patroon een betere foutafhandeling, waardoor de impact wordt verminderd en de stabiliteit van het systeem wordt verbeterd.
- Microservices Pattern
    - Ons project is gebaseerd op de Microservices Pattern, waarbij de applicatie is opgedeeld is in verschillende onafhankelijke en samenwerkende microservices. Elke microservice is gericht op een specifieke functionaliteit. Deze architectuur verbetert de flexibiliteit en schaalbaarheid van het systeem, omdat elke microservice zijn eigen verantwoordelijkheid heeft en onafhankelijk kan werken. De samenwerking tussen microservices wordt mogelijk gemaakt door API's en communicatie via messaging.

SAGA BESCHRIJVING:

1. **Validatie van gebruiker:** Bij het aanmaken van een reservatie wordt als eerste stap van de saga de user gevalideerd. Reservation status wordt gezet op “VALIDATING USER” Het verstuurt via RabbitMQ een bericht naar de UserService die luistert naar een bericht voor het controleren van de gebruiker. Als de gebruiker kan worden gevonden met een bestaand e-mailadres, reageert het met een bericht (Object) waar een boolean (true) mee in wordt gegeven dat de user geldig is. Anders wordt er “false” teruggegeven (User niet geldig/bestaand in ons systeem).
2. **Reservering van auto:** Deze saga gaat verder na de validatiestap van de gebruiker. Als de gebruiker niet geldig is, wordt er een e-mailmelding verzonden waarin wordt aangegeven dat de gebruiker niet bestaat in het systeem. Als de gebruiker geldig is, gaat het verder met het reserveren van de auto door een reserveringscommand te verzenden waar de car-service naar luistert. Het veranderd ook de status van de reservation naar RESERVING_CAR
3. **E-mail Verzenden:** Als de auto beschikbaar is en er zijn geen overlappende reserveringen, gaat het door met het bevestigen van de reservering en wordt de eigenaar op de hoogte gebracht maar ook de gebruiker word geinformeerd dat zijn reservatie moet worden bevestigd door de owner. Na dit stopt de flow even, wachtend op de eigenaars bevestiging. Nu als er sprake is van een dubbele boeking of dat de car niet beschikbaar op het platform (Permanent of tijdelijk), wordt de gebruiker op de hoogte gebracht van het probleem. Reservatie status word ook verandert naar DOUBLE_BOOKING of NOTAVAILABLE in dit geval.
4. **Bevestigen van reservering:** Een API call komt binnen van de iemand die de reservatie wil bevestigen. De saga gaat dan verder door de bevestiging te gaan behandelen. Het stuurt een bevestigingsreserveringscommand naar car-service als de reservering zich in de juiste staat bevindt, of een e-mailmelding als de reservering niet kan worden bevestigd. Car-service zal dan 2 dingen controleren en dat is namelijk dat de gegeven auto bestaat en dat diegene die de API call maakt, ook daadwerkelijk de eigenaar is van de auto.
5. **Facturering van de gebruiker:** Deze saga gaat over de reactie van de eigenaar op een reserveringsverzoek. Als de eigenaar bevestigt, gaat het door met het factureren van de gebruiker. Billing-service krijgt dit binnen en gaat een nieuwe rekening opstellen en het bedrag berekenen (Prijs is per dag dus prijs maal aantal dagen). Als de eigenaar weigert of niet de werkelijke eigenaar is, wordt er een passende e-mailmelding verzonden.
6. **Reservering afronden:** Dit is de laatste saga die wordt uitgevoerd in dit process nadat de factuur is aangemaakt. Het markeert de reservering als voltooid en stuurt een e-mailmelding naar de gebruiker met de factuurgegevens.

Zie hieronder een mooie tekening van de flow van dit proces.


Gebruiker Validatie
Gebruiker is geldig
Gebruiker is niet geldig
Reserveer auto
Verstuur email dat de gebruiker niet geldig is.
Is de auto available en zijn er geen overlapping reservaties?
Ja
Nee
Wachten op confirmatie van de eigenaar.
Verstuur een double booking of not available email naar de gebruiker
Is de reservatie geconfirmeerd?
Yes
Geweigerd
Factureer gebruiker
Verstuur email naar gebruiker
Mark de reservatie als completed en verstuur billing details naar de gebruiker
Made with Whimsical