PSP(Chargebee) Integration High-Level System Architecture

Backend: .NET Core for microservices.
Database: Azure Cosmos DB with MongoDB API. (Azure Cosmos DB provides compatibility with the MongoDB API, which means you can use existing MongoDB libraries, drivers, and tools to interact with Azure Cosmos DB.)
API Gateway: Azure API Management. Can be used together with Azure API Gateway.
Message Queue: Azure Service Bus. (Apache Kafka is also a nice option) (Azure Service Bus would likely be the best fit but Azure Event Hubs are an option too especially when we need to quickly accept events (millions) from end-users.)
Authentication: OAuth or JWT.
Containerization: Docker (if needed for local development or deployment). Azure Kubernetes Service for orchestration
API Documentation: Azure APIM also provides API documentation which we can import via OpenAPI specification
Storage for sensitive information: Azure Key Vault. Secure all services inside a VNET and communication is primarily via APIM (Azure API Management). Also, leverage *Azure Managed Identity* to completely eliminate handling of secure strings where possible.
Monitoring: Use tools like Prometheus, Azure Application Insights and Grafana for monitoring microservices and infrastructure.
Email Infrastructure: SendGrid
Payment system: Chargebee as PSP

High Availability and Disaster Recovery Plan
High Availability:
- Utilize Azure Availability Sets for VMs and AKS for microservices to ensure redundancy. (Deploy resources across multiple zones)
- Microservices can be deployed and scaled independently, allowing for optimal resource allocation and handling increased load. (Autoscaling configured for microservices based on demand.)
- Docker for easy deployment and scaling.
- Container Orchestration (Azure Kubernetes Service - AKS):
- Deploy microservices as containers for scalability and manageability.

Disaster Recovery:
- Regularly backup Azure Cosmos DB to Azure Blob Storage for quick recovery in case of data loss.
- Use Azure Site Recovery for VM-level disaster recovery.
- Implement geographically distributed data centers for additional redundancy.
- Runbook Documentation: Develop runbooks with detailed procedures for recovery in case of a disaster.

Logging and Monitoring (Azure Application Insights):
- Use Azure Application Insights for real-time monitoring microservice performance, troubleshooting and logs. Splunk, New Relic, Datadog, ELK (Elasticsearch, Logstash, Kibana) Stack (for centralized logging, analyzing user actions, and troubleshooting issues) , Prometheus, Grafana (Prometheus and Grafana for monitoring system health, resource usage, and performance metrics) can also be options

API Gateway (Azure API Management):

- Azure API Gateway to manage and route requests. Acts as a single entry point for clients.
- Responsible for authentication, rate limiting, request validation, and routing to appropriate microservices.

Load Balancer (Azure Load Balancer) (not a must):

- Azure Load Balancer for distributing incoming traffic across multiple instances of microservices for scalability and high availability. It can reroute traffic away from unhealthy instances, preventing them from affecting overall system performance. NGINX can be used as well.

** The combination of an API Gateway and Load Balancer allows for both vertical and horizontal scalability, ensuring the system can handle increased loads efficiently. API Gateway enhances security by handling authentication and authorization, while the Load Balancer contributes to system resilience and fault tolerance. Separating concerns between API management and traffic distribution provides a modular and more easily maintainable architecture.Centralized monitoring and logging through the API Gateway help in tracking and diagnosing issues, while the Load Balancer ensures even distribution of traffic for optimal performance. *Azure API Management alone may be sufficient, especially if the APIs are not distributed across multiple backend servers.*

Security Risks and Mitigations

Data Breach:

- Implement encryption at rest, use secure connections (HTTPS) by ensuring Azure App Service is configured to use a valid SSL/TLS certificate, and regularly update security patches.

Unauthorized Access:

- Implement OAuth or JWT for authentication, RBAC for microservices access (authorization), and secure API key management.

Secure Key Management:

- Store and manage sensitive information such as API keys and secrets in Azure Key Vault. (managed identity)

Denial of Service (DoS) Attacks:

- Utilize Azure DDoS Protection.
- Implement rate limiting and throttling (via API Gateway/ APIM).

Webhook Security:

- Use secure HTTPS for webhook communication.
- Validate and verify incoming webhook payloads from Chargebee or any other 3rd party.

Anti Fraud - block fraudsters from signing up (without knowing who they are):

- ChargeBee is detecting fraudulent transactions via IP addresses and is collecting these IP's via Hosted Pages & API Users. Chargebee monitors all transactions and makes note of the IP addresses. This system flags a customer as suspicious if one or more of their transactions are made from a suspicious IP address.

- Chargebee provides the option to block prepaid credit cards. Or virtual cards

Payment Failure Recovery:

- 'Accounts Receivable' product from Chargebee. It's a customizable follow up flow after payments fail. Proactively alert customers on card expiries and missing payment information to avoid disrupting their service and your recurring revenue.

Idempotency:

- Design microservices and event processing in an idempotent manner to handle potential duplicate events. (Save RequestId/eventid/idempotencyKey coming from Chargebee and check whether it's processed already or not. If it's a duplicate event, then simply ignore it and log it to Splunk)

Ordering or messages in the queue:

- Use sessionId to make sure ordering of messages in an Azure Service Bus queue or topic. Imagine a case like this: 2 events are published to the Service Bus queue, UserUpdated & UserUpdated. The first one has Name = "Ece" and the other one has Name = "Ece**m**". If the "Ece**m**" one is processed before the "Ece" for a reason, the latest state in the database will hold the "Ece" which is wrong

Testing:

- Comprehensive testing, including unit tests, integration tests, and security testing.
- Specflow tests
- Perform Canary Testing (before official launch): select a group of clients to participate in testing new features, guaranteeing that the new API aligns with the requirements and operates smoothly.
- Penetration Testing: Periodic testing to discover and address potential security weaknesses.

Other security best practices:

- Configure API Gateway to authenticate requests using *Azure Managed Identity*. This ensures that only authorized services can access the gateway.
- Register each microservice as an Azure Active Directory App. This allows us to configure access permissions and roles for each service.
- For each of your microservices enable Managed Identity on the respective Azure services.
- Ensure that the Blob Storage Service has the necessary permissions to read and write to Azure Blob Storage using Managed Identity.

Data Model

The data model can be designed to represent entities such as Companies (Clients), Users, Subscriptions, and Licenses. Below is a simplified representation of the data model using a document-oriented approach for MongoDB:

Company Collection:

```
{
  "_id": ObjectId("..."),
  "createdAt": ISODate("..."),
  "updatedAt": ISODate("..."),
  "name": "Company Name",
  "email": "company@email.com",
  "expiresAt": ISODate("..."),
  "featurePackage": "Enterprise",
  "contractualUsers": 5
}
```

User collection

```
{
  "_id": ObjectId("..."),
  "createdAt": ISODate("..."),
  "updatedAt": ISODate("..."),
  "name": "User Name",
  "email": "user@email.com",
  "hash": "user_unique_hash",
  "company": ObjectId("...")  // Reference to the Company document
}
```

PaymentOrder Collection (Part of OrderService):

```
{
  "payment_order_id": ObjectId("..."), // idempotencyKey (PK)
  "createdAt": ISODate("..."),
  "updatedAt": ISODate("..."),
  "userId": ObjectId("..."),  // Reference to the User document
  "package": "Enterprise",
  "quantity": 5,
  "paymentStatus": "success", //not_started,executing, success, failed
  "expiryDate": ISODate("..."),
   "amount": "100.0"
}
```

License Collection (Part of License Service):

```
{
  "license_id": ObjectId("..."),
  "createdAt": ISODate("..."),
  "updatedAt": ISODate("..."),
  "orderId": ObjectId("..."),  // Reference to the Order document
  "userId": ObjectId("..."),  // Reference to the User document
  "expirationDate": ISODate("..."),
  "status": "Active"
}
```

Note: We can embed the licenses directly within the Company document as an array. Each license could be represented as an object with relevant details, such as license type, expiration date, etc.

Pros:

Simplicity in retrieval as licenses are directly associated with the company.

All data is stored in a single document, reducing the need for multiple queries.

Cons:

Document size limitations in MongoDB could become a concern if the number of licenses grows significantly.

Updating licenses might require updating the entire Company document.

Instead I would create a separate collection for licenses and establish a relationship with the User entity (Company entity is also possible). Each license document references the ID of the associated Company or User.
Pros:

Allows for more flexibility in managing and updating licenses independently.
Can handle a larger number of licenses without hitting document size limitations.

Cons:

Requires additional queries to retrieve license information when needed.

How Microservices Look like

User Service (Microservice 1):

- Manages user and company-related operations. Update user’s name etc.
- Stores user and company information in MongoDB/CosmosDB.
- If it’s an existing user, the service checks whether the max license amount is exceeded by checking the license quantity in the database
- When a new license is created or updated, UserService updates the license quantity

Order Service (Microservice 2):

- Handles order-related logic.
- When a new order is created, it creates a new PaymentOrder record in MongoDB/CosmosDB
- Communicates with PaymentService via Service Bus when a new order is generated

Payment Service (Microservice 3):

- Integrates with Chargebee for payment processing.
- Handles webhooks from Chargebee to update PaymentStatus in PaymentOrder table.
- Manages payment-related operations.
- The initial status is NOT_STARTED. When the payment service sends the payment order to the Chargebee, the status changes to EXECUTING. The payment service updates the status to SUCCESS or FAILED based on the response from Chargebee.

License Service (Microservice 4):

- License Allocation: Tracks and manages the allocation of licenses to clients (companies).
- Expiration Management: Monitors license expiration dates and takes appropriate actions (e.g., sending notifications, updating the status).
- Usage Tracking: Keeps track of the number of licenses purchased by a client and their current usage.
- License Information Storage: Stores license-related information, including expiration dates, usage, and other relevant details, in the MongoDB database. But the license itself is saved to Blob storage.
- Communicates with the User Service to update license information based on order changes (e.g., an increase in the number of licensed users, allocate or deallocate licenses).

A sample flow (please check Chargebee_Integration.pdf to see the detailed diagram);
Having both Azure Gateway and APIM helped to centralise cross-cutting concerns such as authentication, SSL termination, and load balancing, rate limiting. Depending on the permission model, configure either a key vault access policy or Azure RBAC access for an API Management managed identity. The APIM service performs its security checks, and forwards valid requests to the downstream services in the AKS cluster. All services are secured inside a VNET.

User Registration/Login:
- Users register on the website or log in with existing credentials.
- Upon successful authentication, the website obtains an API key or OAuth token.

API Key or Token Inclusion:
- The website includes the API key or OAuth token in the headers of requests to the API Gateway. (save those API keys to keyvault)

API Gateway Authentication:
- The API Gateway verifies the API key or OAuth token.
- For user-specific actions, it may validate against the user's credentials stored in the User Service.

Authorization Check:
- After authentication, the API Gateway checks the user's role and subscription level.
- It ensures that the user has the necessary permissions to perform the requested action.

Validated Request Forwarding:
- If authentication and authorization are successful, the API Gateway forwards the validated request to the appropriate microservice, User Service in this case.

-> User service checks whether the entered license quantity is valid (not exceeding 10) by querying the related Company table. If it’s a new subscription then there’s no need to check the database as the max amount is always 10.
Note: I assume there are other services for Pricing and Product. And the total amount of selected packages is calculated by checking the Pricing table.
-> User service publishes to package_selected topic with details like total amount, selected package, quantity, expiry date,userId
-> Order service subscribed to package_selected topic. And created a new PaymentOrder record;
```
{
  "order_id": // idempotencykey
  "userId": ObjectId("..."),
  "package": "Enterprise",
  "quantity": 5
  "paymentStatus": "not_started"
  "expiryDate":
   "amount": "100.0"
}
```
-> Order service publishes to order_created topic with all these details and Payment Service receives this information. Upon receiving the payment order information, the payment service sends a payment registration request to Chargebee. This registration request contains relevant payment details such as the amount, currency, expiration date of the

payment request, and the redirect URL. To ensure unique registration and prevent duplication, a UUID field is included. This UUID corresponds to the ID of the payment order.

In the 2nd step, Chargebee returns a token to the payment service, which serves as a unique identifier for the payment registration on Chargebee side. This token enables later examination of the payment registration and payment execution status. **Eg: Use non-recurring payment methods used to do recurring payments.**

- **When the initial payment is done via iDeal and completed successfully, we store the transaction ID (token).**
- **When we upload the payment information to buckaroo after the billing run, we use that transaction ID (token) as OriginalTransaction.**
- **Buckaroo will then do a new transaction of type SEPA Direct Debit, using the IBAN of the original iDeal transaction.**

The payment service stores the token in the database before initiating the call to the Chargebee-hosted payment page. Once the token is saved, the client displays the Chargebee-hosted payment page.
Chargebee handles the collection of sensitive payment information, ensuring it never reaches our payment system. The hosted payment page typically requires two pieces of information:

- The token received in step 3: Chargebee's JS code provides this token to retrieve detailed information about the payment request from Chargebee's backend. One crucial piece of information is the amount to be collected.
- The redirect URL: This is the web page URL that is called upon completion of the payment. When Chargebee's JS completes the payment process, it redirects the browser to the specified redirect URL.

The user enters the payment information, including the credit card number, cardholder's name, expiration date, and other relevant details, on the Chargebee's webpage. After filling in the necessary information, the user clicks the pay button. Then, Chargebee initiates the payment processing procedure. In an asynchronous way, Chargebee communicates the payment status to the payment service through a webhook. During the initial setup with the Chargebee, a URL on the payment system's side is registered as the webhook. When payment events are sent to the payment system through the webhook, the payment system extracts the payment status information and updates the "payment_order_status" field in the PaymentOrder table.
-> If the payment is successful, then Payment Service publishes a payment_successful topic. License service subscribed to this topic receives the information and generates the license. The service stores the license in Azure blob storage and then sends notification to the user using SendGrid. Then user service updates the license quantity information in the Company table.

Note: I preferred using Pub/sub mechanism, because I assumed there can be multiple services to consume the same message. For instance Payment events are published to Azure Service Bus and can subsequently be consumed by various services, including the payment system itself, an analytics service, and a billing service. This design enables the seamless distribution of payment-related information across multiple services, ensuring that each service can perform its specific tasks and process the payment events accordingly.

** Additional consideration: There should be a nightly-scheduled AzureWebjob to check whether there are any expiring licenses by filtering Mongodb. This can be a logic app (add a schedule and runbook to run every midnight) that puts a message in a queue called scheduled-jobs and the webjob queries MongoDB/CosmosDB to find out the licenses that are about to expire, then sends an email to those customers. If there are any licenses to be ended on the day the job runs, then the job needs to update the status of those licenses as Inactive by updating the right record in Mongodb and inform the customer by sending a notification.

Another way is to use a timer function:
In the License Service, we can create a Timer Trigger Function. This function will be triggered at regular intervals to check for licenses about to expire. We can use this function to initiate the notification process.

```
public static class LicenseExpirationCheckFunction
{
   [FunctionName("LicenseExpirationCheck")]
   public static void Run([TimerTrigger("0 0 0 * * *")] TimerInfo myTimer, ILogger log)
   {
      // This function runs every day at midnight (0 0 0 * * *)
      // Implement logic to check for licenses about to expire

      // Trigger the process to notify users about expiring licenses
      NotifyUsersAboutExpiringLicenses();
   }

   private static void NotifyUsersAboutExpiringLicenses()
   {
      // Implement the logic to get licenses about to expire
      // Trigger the Notification Service to send notifications
   }
}
```

Once the LicenseExpirationCheckFunction identifies licenses about to expire, it can trigger the Notification Service. This can be achieved by calling an HTTP endpoint exposed by the Notification Service or by using a message queue. The NotifyUsers function is triggered by a message in the "notificationqueue".

```
public static class NotificationService
{
   [FunctionName("NotifyUsers")]
   public static void Run([QueueTrigger("notificationqueue")] string message, ILogger log)
   {
      // Implement logic to send notifications to users
      log.LogInformation($"Sending notification: {message}");
   }
}
```

How to handle failed payments

- Retry queue: retryable errors such as transient errors are routed to a retry queue.
- Dead letter queue: if a message fails repeatedly, it eventually lands in the dead letter queue. A dead letter queue is useful for debugging and isolating problematic messages for inspection to determine why they were not processed successfully.

Check whether the failure is retryable.

- Retryable failures are routed to a retry queue.
- For non-retryable failures such as invalid input, errors are stored in a database.

The payment system consumes events from the retry queue and retries failed payment transactions.

- If the payment transaction fails again:
- If the retry count doesn't exceed the threshold, the event is routed to the retry queue.
- If the retry count exceeds the threshold, the event is put in the dead letter queue. Those failed events might need to be investigated.

Another concern is to achieve **exactly-once delivery**. One of the most critical issues that a payment system can encounter is double-charging a customer. By combining retry mechanisms for at-least-once execution and implementing idempotency checks for **at-most-once execution**, we can create a robust and reliable payment system that ensures the execution of payment orders exactly once, mitigating the risk of double-charging customers.

Here are some common retry strategies:

- Immediate retry: The client promptly resends the request after a failure occurs.
- Fixed intervals: A fixed amount of time is waited between the failed payment and subsequent retry attempts.
- Incremental intervals: The client initially waits for a short period before the first retry and then gradually increases the waiting time for subsequent retries.
- Exponential backoff: The waiting time between retries is doubled after each failed attempt. For instance, if the request fails the first time, a retry is attempted after 1 second. If it fails again, the next retry is attempted after 2 seconds, and so on.
- Cancel: The client has the option to cancel the request, especially when the failure is permanent or further retries are unlikely to succeed.

One potential problem with retries is the possibility of double payments.

<u>Scenario: What if a customer clicks the "pay" button quickly twice?</u>

When a user clicks the "pay" button, an idempotency key is included in the HTTP request sent to the payment system. In the case of a second request, it is considered a retry since the payment system has already encountered the idempotency key (payment_order_id). By including the previously specified idempotency key in the request header, the payment system responds by providing the latest status of the previous request. This approach ensures that duplicate requests or retries are effectively handled and prevents unintended consequences that may arise from multiple submissions of the same payment. If multiple concurrent requests are detected with the same idempotency key, only one request is processed and the others receive the “429 Too Many Requests” status code.

<u>Scenario: The payment is successfully processed by Chargebee, but the response fails to reach our payment system due to network errors. Then the user clicks the “pay” again.</u>
if the payment is successfully processed by Chargebee, but the response fails to reach our payment system due to network errors, and subsequently the user clicks the "pay" button again, the following scenario can occur:

- The second "pay" request is sent to the payment system, unaware that the initial payment was actually successful.

At this point, the payment system needs to handle the duplicate request and ensure that the duplicate payment is not processed twice, preventing any unintended consequences or duplicate charges.
To address this situation, the payment system can implement mechanisms such as idempotency keys or unique identifiers. By including an idempotency key in the request, the payment system can identify and recognize the duplicate request. It can then determine that the payment has already been successfully processed and respond accordingly, without processing the payment again. This helps maintain data integrity and prevents any duplicate or erroneous transactions caused by network errors or user interactions.