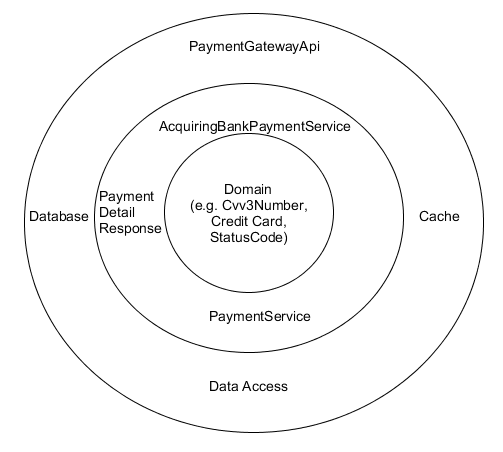
**Payment Gateway Api | Checkout Challenge | Daniel Botero Correa**

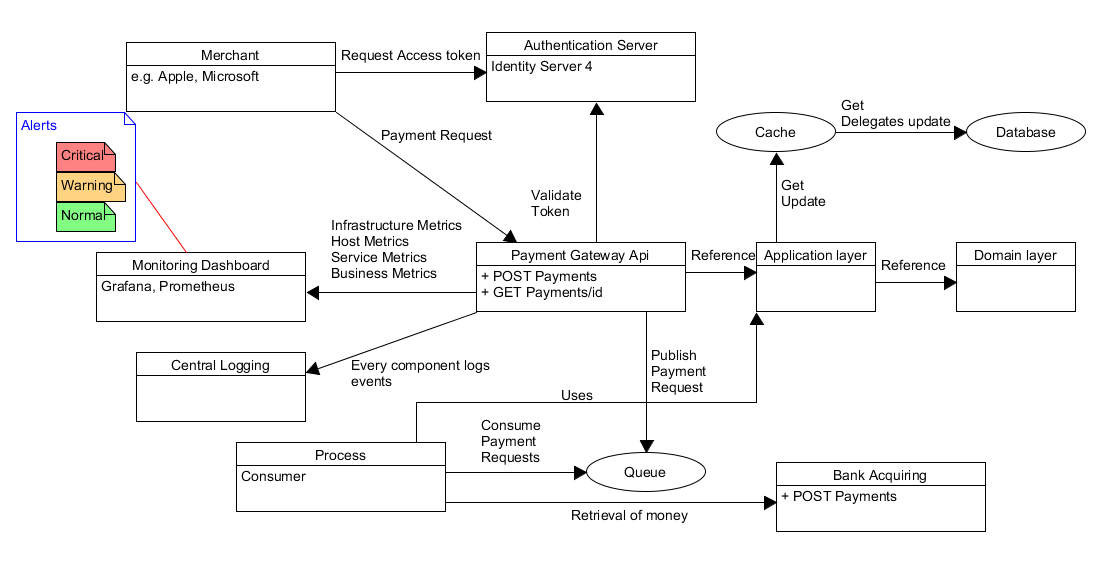
**Summary**

**Application Architecture**

I designed the application using a onion architecture style



**High level design of the system**



This solution leverage the power of interfaces to decoupled the application from concrete implementations, it is loosely coupled to aspects such as:

**Authentication/Authorization Mechanism**: This is dealt within the PaymentGateway.Api, I use IdentityServer4 to enable merchant to Authenticate with and be Authorized by when calling api/Payments endpoints. IdentityServer4 can be configured to use any kind of Oauth2-based authentication.

**Logging**: I used .Net Core abstractions for logging from the ground up. It enables me to switch my logging provider from one to another without affecting my Business Logic and write to different logging services only by changing the PaymentGateway.Api ‘s configuration file. I used Serilog as Logging library.

**Persistence**: I used the Repository Pattern to decouple any data access detail from my business logic, that helps me to build a more flexible architecture. For Intance, I used a InMemoryDatabase that can be easily switched for a MySql database.

**Caching**: The cache implement the same interfaces that the persistence (for instance, IPaymentRepository). This allowed me to create a Proxy Pattern and intercept any call to the persistence while checking whether a value is present in the cache.

**Consumer/Producer**: I created an abstraction for Consumer/Producer of messages. This allow me to switch from my current implementation to another without changing my business logic.

**Encryption**: I used a really basic algorithm to encrypt/decrypt CreditCard details but it can be switched by any concrete implementation as long as it respects the interface for this purpose.

**Applications and Tools:**

**IdentityServer4**: Authentication/Authorization server.

**Prometheus**: Software application used for event monitoring and alerting

**Grafana**: Analytics and interactive visualization software, Prometheous is the data source for this application.

**Graylog**: Centralized log management system

**Swagger**: Open-source software framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful web services.

**Payment Gateway Application Characteristics**

**Resilient**: I have implemented the Timeout and Retry pattern so the system is able to process all the payment requests even if the Acquiring bank Api is down.

**Fast**: Payment requests are validated and then queued in an InMemoryQueue (which can be replaced by a real queue). This allows me to receive thousands of requests in parallel as I give a fast feedback to the user. Then a consumer treats the requests making my system eventual consistent.

Any retrieve payment request is cached so the next time we want to access it, the system doesn’t have to query the database but the caching system. If a payment request is updated after Acquiring bank feedback, the payment request is removed from the cache.

**Scalable**: Assuming I switch to a real queue and real database, I can spin up multiple instances of my services which will make the system scalable in times of high demand. This can be done using AWS services for instance.

**Testable**: As my application is loosely coupled to the concrete implementations, I can tests any part of my system independently.

**Easy to debug**: As my applications use logs to keep track of most parts of the system, we can easily track what happened during execution.

**Metrics**: Send Business metrics so I know if the per-hour/daily rates are respected.

**HATEOAS:** (Hypermedia as the Engine of Application State)**:** The application provide information dynamically through hypermedia so clients can be almost decoupled from it as well as making it more flexible to change.

**Maintainable:** Most of the above make the application maintainable

**Technology**

The application has been developed using .Net Core Framework and C#.

**In more details:**

**Logging**: Logging purposes I used Serilog (File, Console, Graylog)

**Metrics**: Prometheous and Grafana

**Queue**: InMemory BlockingCollection

**Cache**: MemoryCache

**Authorization/Authentication Server**: IdentityServer4

**Api interface**: Swagger

**Testing**: NUnit, Moq

**Workflows**

**Receive payment request**

A merchant has to authenticate with our **authorization server**, get an access token and send it as Bearer token within the payment request. A payment request looks like:

POST /api/Payments HTTP/1.1

Host: localhost:53746

Content-Type: application/json

Authorization: Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6ImNabFQ5MnRKY3hDcHItYVY3NkgzS3ciLCJ0eXAiOiJhdCtqd3QifQ.eyJuYmYiOjE1ODQ2OTk1NTQsImV4cCI6MTU4NDcwMzE1NCwiaXNzIjoiaHR0cHM6Ly9sb2NhbGhvc3Q6NTAwMiIsImF1ZCI6IlBheW1lbnRHYXRld2F5QXBpIiwiY2xpZW50X2lkIjoiQXBwbGUiLCJzY29wZSI6WyJQYXltZW50R2F0ZXdheUFwaSJdfQ.LC3cDJQJ5NOyAxbprT8muYHpT6zq2JSR9ewDGygKXtCLWijlF2zqZIx6ytMotOgZ74XnCNpUfsG6XbcsggrbCdfhYIv-U6HhBmN9x\_1KC26RHYCkwrQobRwbh\_tFud8Yyoj3VO-BdncEo\_P73TmPmoPtUhQ8WwtJFaKcTYTloPBl7rHwGtVGoKo-tWPUsRLhv\_\_7SJtIyYCPXty5Mc8MoWMXAkGHRnPvGOzpawZLX88iNIjZ5ETHOJrlOMmWLOGTqKjdOeQj89Wi8cdtCy0uj7A3SNU1fru5vWqey-rc2odFCTco4rSWsPUPFtXZwgILHW2GY0bg-4O50cMEty0LQA

Cache-Control: no-cache

Postman-Token: a285e4bb-dc2a-da40-0221-6e84f4253319

{

"Merchant":{

"Name" : "Apple"

},

"CreditCard":{

"Number":"1234 5678 9101 1213",

"ExpirationDate" : "2025-04-23T00:00:00.000Z",

"Cvv": 123,

"HolderName": "Daniel Botero Correa"

},

"Amount":125,

"Currency" : "EUR"

}

The request is parsed and validated with rules that the Business provides us.

If the requests succeed, the api response is:

{

"result": {

"id": "a6956972-8314-4cb0-b03e-84bf21ff915a"

},

"\_links": [

{

"self": {

"href": "https://localhost:44346/api/Payments/a6956972-8314-4cb0-b03e-84bf21ff915a"

}

}

]

}

If the request has errors, the api response might be:

{

"error\_type": "request\_invalid",

"error\_codes": [

"CardNumberMustBeNumeric16Digits",

"CvvMustContain3Numbers",

"ExpiryDateHasExpired",

"HolderNotEmpty",

"NonNegativeAmountViolation",

"MerchantNotExists",

"CurrencyNotExists"

]

}

I have structured my api based on checkout bancontact doc: <https://docs.checkout.com/docs/bancontact>

**Retrieve payment request**

A merchant, after authentication and using the link provided in the Payments Received use case **(**<https://localhost:44346/api/Payments/a6956972-8314-4cb0-b03e-84bf21ff915a>) can request the payment to check out the state of the payment request. The request looks like:

GET /api/Payments/1ad61861-568e-493f-880d-cec18f324a05 HTTP/1.1

Host: localhost:44346

Content-Type: application/json

Authorization: Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6ImNabFQ5MnRKY3hDcHItYVY3NkgzS3ciLCJ0eXAiOiJhdCtqd3QifQ.eyJuYmYiOjE1ODQ5Njc1MzUsImV4cCI6MTU4NDk3MTEzNSwiaXNzIjoiaHR0cHM6Ly9sb2NhbGhvc3Q6NTAwMiIsImF1ZCI6IlBheW1lbnRHYXRld2F5QXBpIiwiY2xpZW50X2lkIjoiQXBwbGUiLCJzY29wZSI6WyJQYXltZW50R2F0ZXdheUFwaSJdfQ.UgEowDpN5U1RBOwZE9AKO6OGyqmL8-qfAXhnd962MPoSLj3Gmnw-7EwUM5W-5ZaYwZlsgSde4N9-a2XAYQR3rGhQF1Li-4CwjogeLdAB1\_HN-Y-nWGRE2koBzoQvsvDsj\_3ghjef0gvycCtG3FNW7VVa78d1lPe3s22gXmKQX\_CKNlzfPxE5afMKscjgB85\_SNY1oRFqVLl6IFcKRySuXH6ALyuoYNhaNvQ8YCa7U4XrHn3zleIjzgqIxib9uy5WLpgf8xnSyxLuZ\_MRww7Md5HchQeFtJXUaq3U3yV78GJ0eNIDmppLiOBlpIYEu5KAAq3JH0ngkasORM8bcYQucw

Cache-Control: no-cache

Postman-Token: c4a1af2a-c868-6c9b-7ce0-25911c152058

The response for this request looks like:

{

"result": {

"id": " a6956972-8314-4cb0-b03e-84bf21ff915a ",

"merchant": {

"name": "Apple"

},

"creditCard": {

"number": "XXXX XXXX XXXX 1213",

"expirationDate": "2025-04-23T00:00:00",

"cvv": 0,

"holderName": "Daniel Botero Correa"

},

"amount": 125,

"currency": "EUR",

"statusCode": "Failure",

"reason": "Customer doesn't have enough money"

},

"\_links": [

{

"self": {

"href": "https://localhost:44346/api/Payments/ a6956972-8314-4cb0-b03e-84bf21ff915a "

}

}

]

}

If the payment request hasn’t been treated by the Acquiring bank, the response would look like:

{

"result": {

"id": " a6956972-8314-4cb0-b03e-84bf21ff915a ",

"merchant": {

"name": "Apple"

},

"creditCard": {

"number": "XXXX XXXX XXXX 1213",

"expirationDate": "2025-04-23T00:00:00",

"cvv": 0,

"holderName": "Daniel Botero Correa"

},

"amount": 125,

"currency": "EUR",

"statusCode": "Pending"

},

"\_links": [

{

"self": {

"href": "https://localhost:44346/api/Payments/ a6956972-8314-4cb0-b03e-84bf21ff915a "

}

}

]

}

**Validate payment request with Acquiring Bank**

A consumer picks up the payment request and send it to the acquiring bank. The following cases could present:

1. Everything goes wrong
   1. Consumer sends the request to the acquiring bank
   2. Request timed out: I create a timeout policy. If a request is timed out, I cancel the request.
   3. If the request fail for some reason (e.g. Timeout, service unavailable), a retry strategy has been set up, after X number of fails the request is send back to the consumer as “faulted”.
   4. Consumer queue up again the request for later treatment.
2. Everything goes ok
   1. Consumer sends the request to the acquiring bank, the Acquiring service receives the response and update the payment using the information returned by the acquiring service.

**Business Metrics**

The business metrics I found nice to have were:

**Time Payments Retrieved:** Average time per request when payment is retrieved

**Time Payments Received:** Average time per request when payment is received

**Payments Received:** Number of payments received

**Payments Retrieved:** Number of payments retrieved

**Payments Received Errors:** Number of errors when payment request received

****