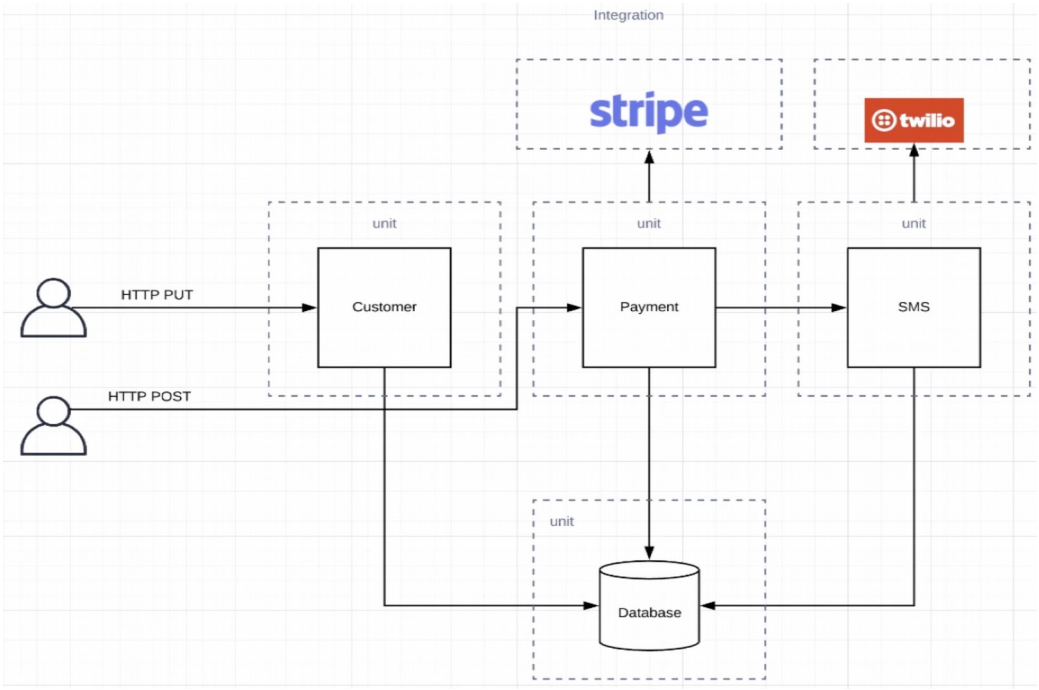
SECTION 06 - INTEGRATION TESTING

Index

- Intro

- In integration testing we test that all the pieces work together whereas the unit test focuses on the unit itself.

- Integration test means that you have the application up and running.



- Mock Stripe service

```
@ConditionalOnProperty(
        value = "stripe.enabled",
        havingValue = "false"
        )
```

- Makes that with integration testing stripe is disabled so that every time we run our application Spring will initialize the mock service being injected instead of StripeService class.

- We can read it as: This bean will be initialized whenever the stripe.enabled property has the value of *false*.

Procedure:

→ From Stripe service:

```
@Service
@ConditionalOnProperty(
        value = "stripe.enabled",
        havingValue = "true"
        )
public class StripeService implements CardPaymentCharger {
```

→ From Stripe mock service:

```
@Service
@ConditionalOnProperty(
        value = "stripe.enabled",
        havingValue = "false"
        )
public class MockStripeService implements CardPaymentCharger {
```

→ From application.properties

```
stripe.enabled=false
```

According to the environment I set stripe.enabled property as I need it.

- PaymentIntegrationTest class

- Add @SpringBootTest annotation on top of the class so whenever I run a test inside an integration test class, Spring will start the entire application.

- If I do:
  → Given:

  → CustomerRegistrationController:

```
@RestController
@RequestMapping("api/v1/customer-registration")
public class CustomerRegistrationController {

    @PutMapping
    public void registerNewCustomer(
            @Valid @RequestBody CustomerRegistrationRequest request) {
        System.out.println(request);
    }
}
```

  → PaymentIntegrationTest:

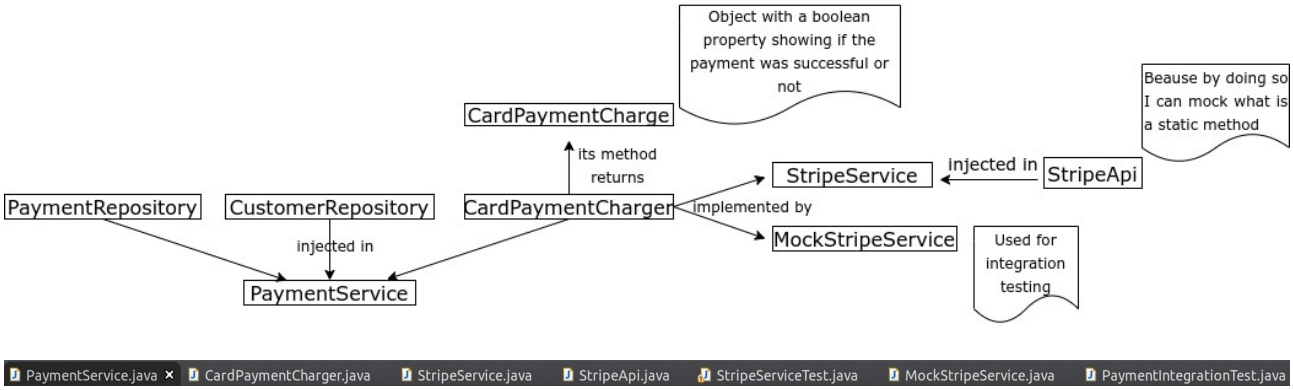
```
@Autowired
private CustomerRegistrationController customerRegistrationController;

@Test
void itShouldCreatePaymentSuccessfully() {
    // given
    UUID customerId = UUID.randomUUID();
    Customer customer = new Customer(customerId, "james", "123");

    customerRegistrationController.registerNewCustomer(...);
```

- By injecting CustomerRegistrationController in the test class I'm not testing the API, instead I'm simply invoking the method directly. What I want to test is when I perform a PUT request to api/v1/customer-registration.

- In itShouldCreatePaymentSuccessfully() test case I "break a rule" by using:

```
@Autowired
private PaymentRepository paymentRepository;
```

  in PaymentIntegrationTest because I don't have any endpoint to get a customer given its id.



- Why is sometimes needed to work with an interface?

In case of **CustomerService** a concrete class is used like **PhoneNumberValidator** and not an interface. In this case when I want to perform unit tests I can mock without any problem the concrete class so when its method called *test* is invoked, it returns true or false. And when the integration test is permormed (**PaymentIntegrationTest**), I will be interested in using **the real PhoneNumberValidator** class. So there is no need to create an interface.

Different is the case of **PaymentService** where an interface is used called **CardPaymentCharger** instead of using directly the **StripeService** class. But why?. Well, if in *PaymentService* there would be *StripeService* (a concrete class), when I run **PaymentIntegrationTest** the original service will be used so I should have to spend money each that I run a test. Because that is ridiculous (and expensive) an interface is used (CardPaymentCharger) in place which is implemented by StripeService and a mock class of that service called **MockStripeService**. This mock class will always return *true*.

So when I run unit tests from **PaymentServiceTest**, I mock *CardPaymentCharger* interface so when I invoke the method *chargeCard* it will return a certain response. Otherwise when I run *PaymentIntegrationTest*, I set up previously Spring Boot so that **MockStripeService** be invoked instead of the real service (StripeService). In this way, StripeService will only be used (for example) in production.

- Testing a static method

The way of testing a static method is by wrapping it inside a class and inject it in the class where the static method was invoked.

Before:

```
public class StripeService implements CardPaymentCharger {
    ...
    Charge charge = Charge.create(requestMap, options);
    ...
```

After:

```
public class StripeService implements CardPaymentCharger {
    ...
    private final StripeApi stripeApi;

    @Autowired
    public StripeService(StripeApi stripeApi) {
        this.stripeApi = stripeApi;
    }
    ...
    Charge charge = stripeApi.create(params, requestOptions);
    ...
```

→ Being StripeApi a service class:

```
@Service
public class StripeApi {
    public Charge create(Map<String, Object> requestMap, RequestOptions options) throws StripeException
{
        return Charge.create(requestMap, options);
    }
}
```

→ From the test class:

```
    @Mock
    private StripeApi stripeApi;

    @InjectMocks
    private StripeService underTest;

    ...

    Charge charge = new Charge();
    charge.setPaid(true);
    given(stripeApi.create(anyMap(), any())).willReturn(charge);
```