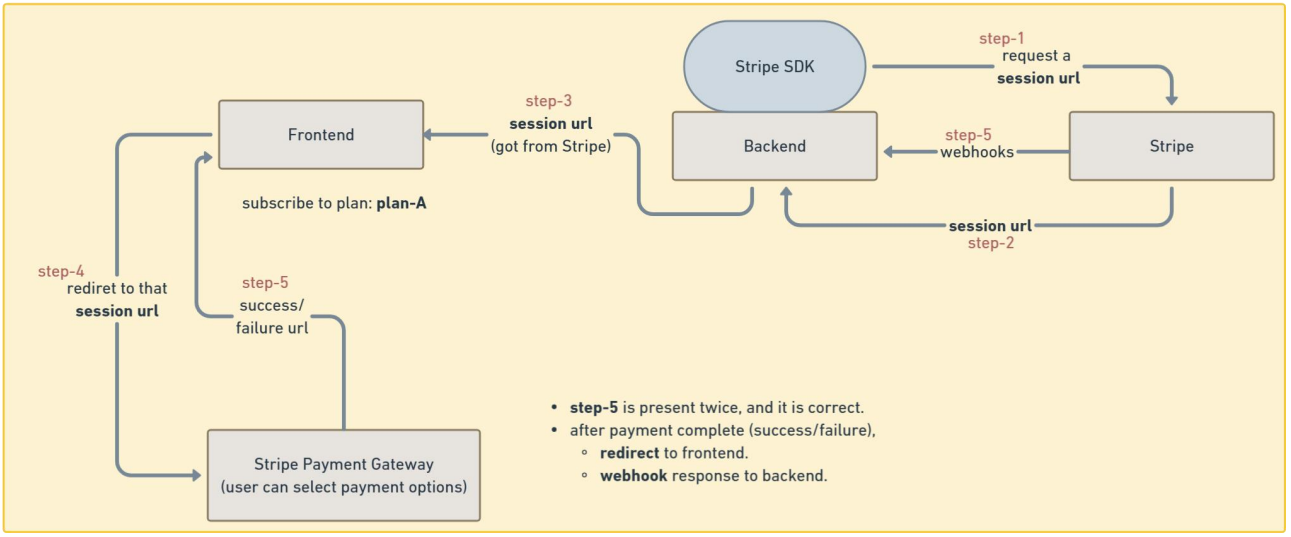
➢ @PostConstruct

- It runs the method (which is being annotated with it) just after the bean is created.
- **init()** method also does the same, but it was there in legacy XML config.
-

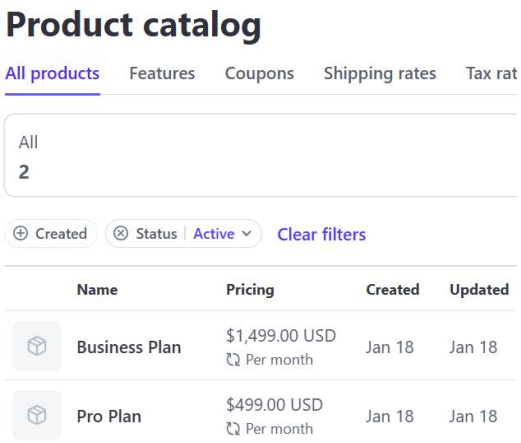
```
@PostConstruct  Alok Ranjan Joshi
public void init() {
    Stripe.apiKey = stripeSecretKey;
}
```

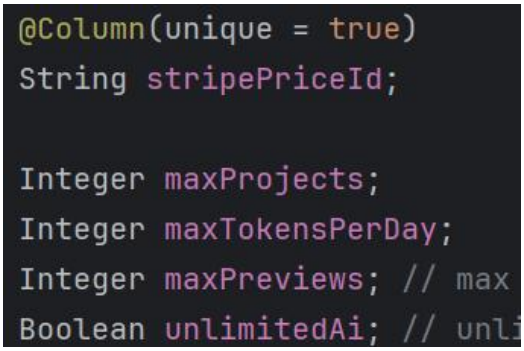
here, the method name can be anything, it is not bounded with **init()**, only the **@PostConstruct** matters here.

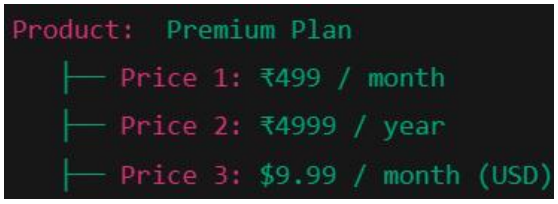
- **Payment Integration Flow**

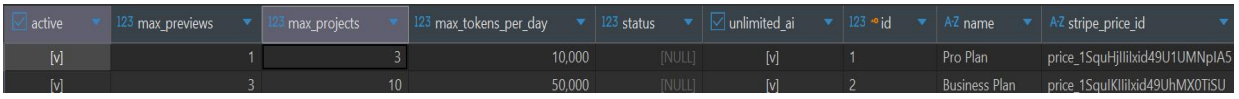


- Why **redirect to frontend** and **webhook for backend** exists?
  - **Redirect** is to make the user know about the status, like success or failure.
    But user can fake it by manually typing the url and will get a free subscription if webhook doesn't exist.
  - **Webhook** on the other hand, sends response to the backend about the status of payment, backend will verify it and stores in the database.
- There is a keyword called **volatile** in ***Java***.
  - `public static volatile String apiKey;` (Stripe class)
  - **volatile** make sure that if one thread changes the variable, then other thread will immediately sees that.
  - When you use **volatile**, all the thread will reference to variable present in main memory directly, not from the **thread local cache**.
- For stripe, you need to set the API key with **Stripe** class. [Stripe related]
  - You will set that **apiKey** variable.
  - ```
    @PostConstruct  Alok Ranjan Joshi
    public void init() {
        Stripe.apiKey = stripeSecretKey;
    }
    ```
    ```

- As we need subscription based payment, like pro plan, normal plan etc, we need to create product in **stripe dashboard**. [Stripe related]
  - 

- Now you need to make the **entity** for **Plan** as well. [Stripe Related]
  - 

  - That **stripePriceId** is there inside the stripe dashboard.
  - Its not the **Product_id**, its **Price_id** (both are different in Stripe Dashboard)
  - And these should be directly appended by database, or you can create a REST end point that can only be accessed by the authorized people.
  - One product (in our case Plan) should have same price, why there can be multiple prices under a single product in Stripe?
    - 

(you can think like this)
  - One **product_id** can have ***multiple*** **price_id**.
  - 

active	max_previews	max_projects	max_tokens_per_day	status	unlimited_ai	id	name	stripe_price_id
[v]	1	3	10,000	[NULL]	[v]	1	Pro Plan	price_1SquHjIlixid49U1UMNpIA5
[v]	3	10	50,000	[NULL]	[v]	2	Business Plan	price_1SquIKIlixid49UhMX0TiSU

  - https://docs.stripe.com/billing/quickstart here you can checkout the Stripe integration docs.

Creating Checkout Session URL

```
var paramsBuilder = SessionCreateParams.builder()
        .addLineItem(
                element: SessionCreateParams.LineItem.builder()
                        .setPrice(plan.getStripePriceId())
                        .setQuantity(1L)
                        .build()
        )
        .setMode(SessionCreateParams.Mode.SUBSCRIPTION)
        .setSubscriptionData(
                new SessionCreateParams.SubscriptionData.Builder()
                        .setBillingMode(SessionCreateParams.SubscriptionData.BillingMode.builder()
                                .setType(SessionCreateParams.SubscriptionData.BillingMode.Type.FLEXIBLE)
                                .build()
                        )
                        .build()
        )
        .setSuccessUrl(frontendUrl + "?success=true&session_id={CHECKOUT_SESSION_ID}")
        .setCancelUrl(frontendUrl + "?cancel.html")
        .putMetadata("user_id", userId.toString())
        .putMetadata("plan_id", plan.getId().toString());
```

- Creates a Stripe Checkout Session to start a payment flow
- Adds one line item using a specific Stripe Price ID (not product) **[addLineItem]**
- Sets the purchase type as a recurring subscription **[setMode]**
  - Ensures Stripe handles automatic recurring billing
- Uses flexible billing mode to allow future plan changes (upgrade/downgrade, proration) **[setSubscriptionData]**
- **Redirects** the user to a Stripe-hosted checkout page
  - On successful payment, redirects back to the frontend with the Checkout Session ID **[setSuccessUrl]**
  - On cancellation, redirects the user back to the frontend cancel page **[setCancelUrl]**
- Attaches metadatas **[putMetadata]**
  - Attaches internal user ID as metadata for payment tracking
  - Attaches internal plan ID as metadata to identify which plan was purchased
  - Allows backend to later verify payment and subscription via *webhooks*
- Keeps payment handling PCI-compliant by using Stripe Checkout.
- Here I am creating a **session *builder***, not ***session***; if you see, I have not called the **build()** method at the end.

```
Redirect  → "Tell the user what happened"
Webhook   → "Tell the system what actually happened"
```

- After that you need to create **session**

```
Session session = Session.create(paramsBuilder.build());
return new CheckoutResponse( checkoutUrl: session.getUrl());
```

  - This **Session.create(…)** method makes an API call to stripe and get a ***session*** object.
  - This **session** object contains the ***url*** that needs to be sent to the frontend.
- One catch is there here:
  - Even if the same user is doing payment, then also inside Stripe, multiple **customers** will be created, because Stripe doesn't know about our customers.

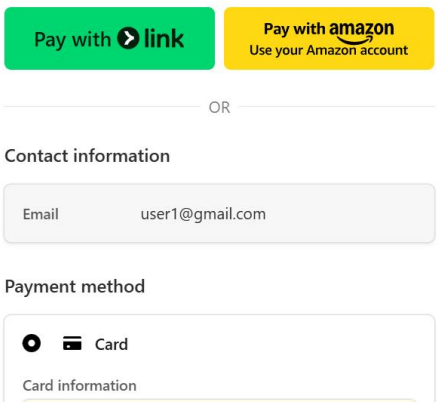
Customer	Email	Primary payment metho
Alok	user1@gmail.com	VISA •••• 4242
Alok	user1@gmail.com	VISA •••• 4242
Alok	user1@gmail.com	VISA •••• 4242

  - Same gmail id (same user), but different customers got created inside Stripe.
- Now to make the Stripe knows if the user is same or not, you need to do the below.

```
String stripeCustomerId = user.getStripeCustomerId();
if(stripeCustomerId == null || stripeCustomerId.isEmpty()) {
    paramsBuilder.setCustomerEmail(user.getUsername());
} else {
    paramsBuilder.setCustomer(stripeCustomerId); // stripe customer id
}

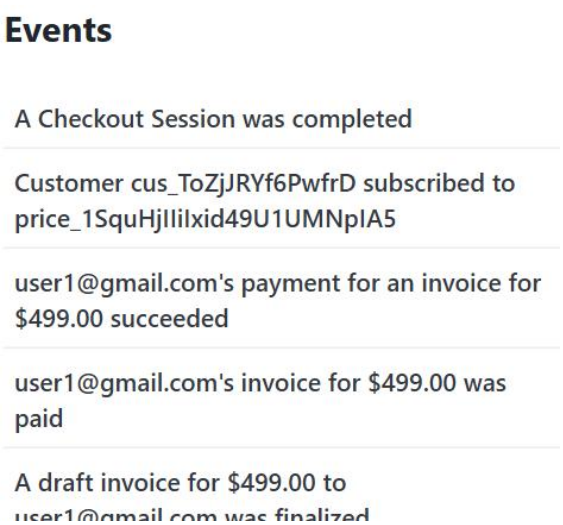
Session session = Session.create(paramsBuilder.build()); /// making API cal
return new CheckoutResponse( checkoutUrl: session.getUrl());
```

  - Now you can see, the email is fixed (if the user is already present), & you cannot edit the email id in Stripe payment page.



- But in database we don't have the customer id present; this will be handled in webhook event handling.

- There are multiple events get triggered



  - Here you can see the customer id and stripe id present (top 2nd event)
  - So, you need to create a **webhook** that listens to this event and you'll get the customer id.
  - And also, one webhook to see the payment status (success/failure)

- [[

  Now Frontend will get this checkout session URL;

  Then Frontend will redirect to that URL;

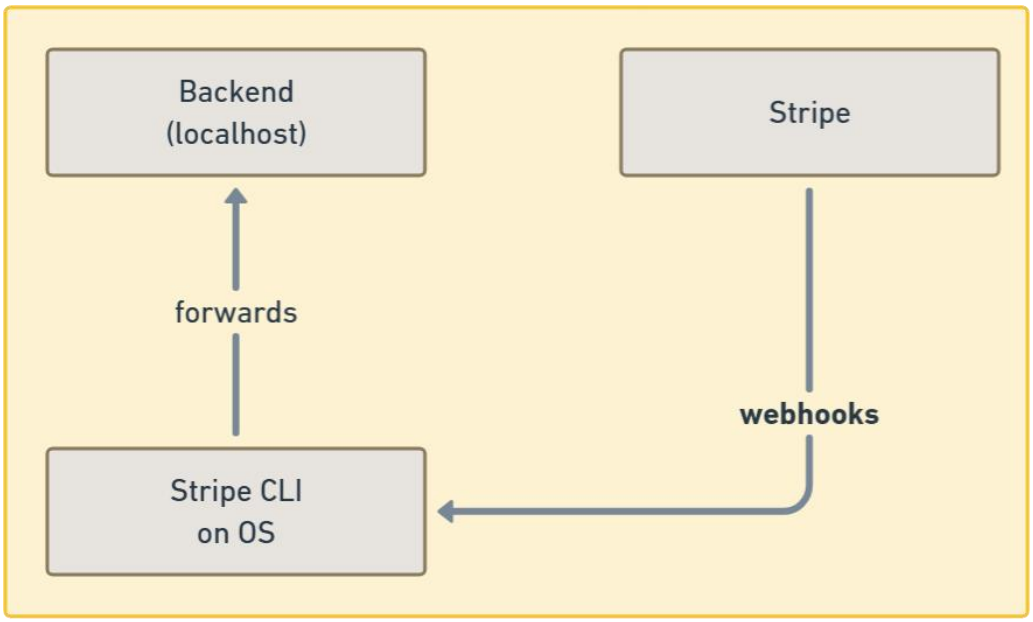
  Then after payment completion, Stripe will respond back to the Redirect URL to frontend & trigger Webhook to send response to the backend.

  --- so we need to create the **webhooks** now ----

  ]]

- Stripe docs for java: https://docs.stripe.com/api/errors?lang=java
- F
- F
- F
- F

Webhooks

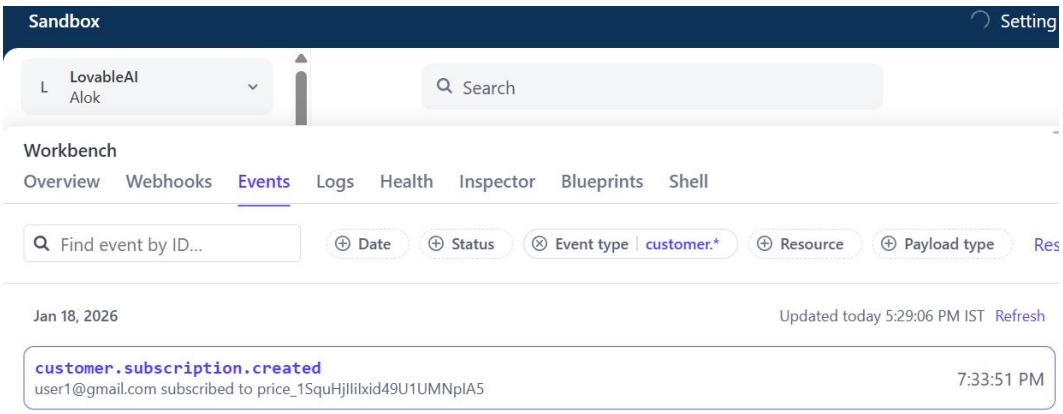
- As our backend is running on *localhost*, Stripe won't be able to send the webhook response to our backend, so we need something to listen to those and forward to localhost.
  - Stripe CLI does this. You need to configure this.
  - Stripe CLI docs: https://docs.stripe.com/stripe-cli/use-cli
  - 

  - When you run the **stripe listen** command in CLI, you'll get a secret called *webhook signing secret*

```
PS C:\WINDOWS\System32> stripe listen --forward-to localhost:8080/webhooks/payment
A newer version of the Stripe CLI is available, please update to: v1.35.0
> Ready! You are using Stripe API Version [2025-12-15.clover]. Your webhook signing secret is
whsec_b5643f423774f00c8e858ad86d4e1e12a5a34f7dda4323cd149274e9bf1c5f6d (^C to quit)
```

- We discussed about **redirect** and **webhooks**, but still ***webhooks*** is not completely secure.
  - Because the webhook endpoint of the backend is a public url, and anyone can send request to that leading to get a free subscription.
  - So, stripe sends a **webhook signing secret** which is being validated at the backend that the request is coming from the Stripe or not.

- You can see in the inspector in the Stripe dashboard page about all the events that was triggered.



- Create an end-point to listen the Stripe webhook.
  - It'll always be a ***post*** end-point.

```
stripe listen --forward-to localhost:8080/api/v1/webhooks/payment
```

    - Here you need to give the proper end-point that handle the webhooks.

```
@PostMapping("/webhooks/payment")  Alok Ranjan Joshi *
public ResponseEntity<String> handlePaymentWebhooks(
        @RequestBody String payload,
        @RequestHeader("Stripe-Signature") String sigHeader
) {
```

    - Webhook handling controller.
    - The header will be having the key **Stripe-Signature**, it contains the signature.

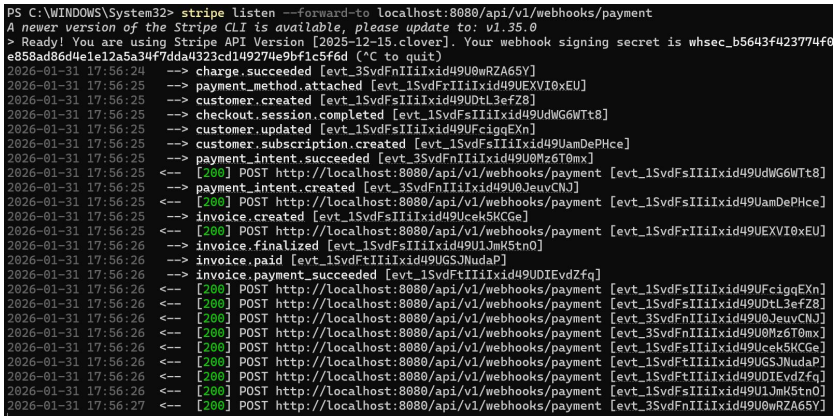
```
Event event = Webhook.constructEvent(payload, sigHeader, webhookSecret);

EventDataObjectDeserializer deserializer = event.getDataObjectDeserializer()
StripeObject stripeObject = null;
```

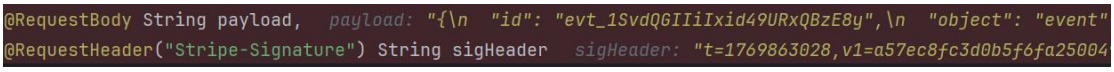
```
stripeObject = deserializer.getObject().get();
```

```
Map<String, String> metadata = new HashMap<>();
if(stripeObject instanceof Session session) {
    metadata = session.getMetadata();
}
```

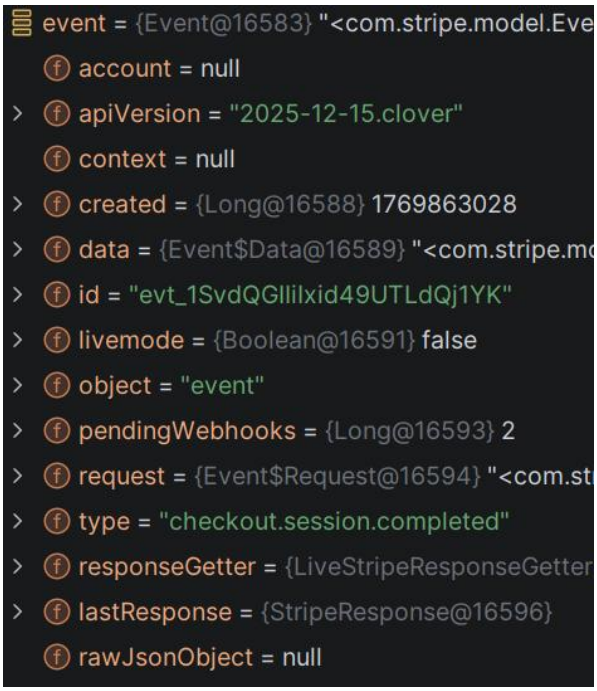
- Now, when you make payment with the url that you get from the checkout response, you'll see the logs in the CLI (stripe listen …etc



- These all are events, *charge.succeeded, payment_method.attached* ….etc etc.
- And for these events, the webhook event handler endpoint will be triggered **once per event**. (here you can see all the POST requests)



- These are the payload and signature header values.



- This is the **event** object that was created using the *secret key, signature header & payload*. You can see ***event type*** is also there.

- After then you you create **EventDataObjectDeserializer** object using that ***event object***,

- It contains a key "**object**" which contains an object of type ***Session*** *(it extends StripeObject class)*

- That ***Session*** object contains all the details like user's registered email in Stripe, the **metadata** that was being attached while creating the session (checkout session)

-

➢ F

- Events of Webhooks
  - checkout.session.completed [
    - This event occurs only once in our case as it is recurring.
    - User will only go to checkout page once and set the auto-pay kind of thing.
    - Then every interval (month/year) stripe will try to deduct by itself.
  - invoice.paid
    - This event is triggered when the payment is done successfully.

```java
public void handleWebhookEvent(String type, StripeObject stripeObject, Map<String, String> metadata) {
    log.info("type = {}", type);

    switch (type) {
        ///  one-time, on checkout completed
        case "checkout.session.completed" -> handleCheckoutSessionCompleted( session: (Session) stripeObject, metadata);
        ///  when user cancels, upgrades or any updates   (------- cancel might not trigger this -------)
        case "customer.subscription.updated" -> handleCustomerSubscriptionUpdated( subscription: (Subscription) stripeObject);
        ///  when subscription ends
        case "customer.subscription.deleted" -> handleCustomerSubscriptionDeleted( subscription: (Subscription) stripeObject);
        ///  when invoice is paid  (due payment or subscription auto-pay is paid successfully)
        case "invoice.paid" -> handleInvoicePaid( invoice: (Invoice) stripeObject);
        ///  when invoice is not paid, mark as PAST_DUE
        case "invoice.payment_failed" -> handleInvoicePaymentFailed( invoice: (Invoice) stripeObject);
        default -> log.debug("Ignoring the event: {}", type);
    }
}
```

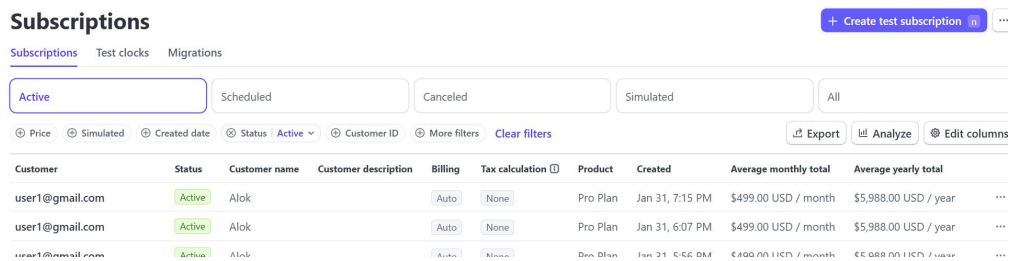
- This is the webhook handler method where these many events are being handled.

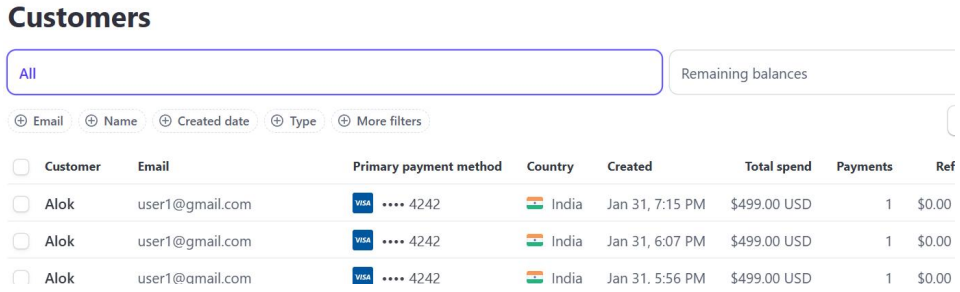
  [check the comments for their purpose]

- In Stripe, **StripeObject** is the parent class of all Stripe objects.

- **handleCheckoutSessionCompleted(session, metadata)**

```
private void handleCheckoutSessionCompleted(
        Session session,
        Map<String, String> metadata) {
}
```

  - On, **checkout.session.completed**, a new subscription will be created, so here we'll set the customer id in our database.
    - It contains all like subscription creation, invoice paid, customer updation ..etc.
  - 

    - These are all the subscriptions.
    - Each subscription is having one **unique id**.
  - 

    - These all are customers (redundant till now, because customer id is not stored in DB yet).
    - Each customer is having one **unique id**.

- F

- F

```
aymentProcessor    : type = customer.subscription.deleted
thFilter           : /api/v1/webhooks/payment
aymentProcessor    : type = invoice.updated
thFilter           : /api/v1/webhooks/payment
thFilter           : /api/v1/webhooks/payment
aymentProcessor    : type = refund.created
```

- F
- F
- F
- F
- F
- F
-