

# Working Group #1

---

**Facilitation Guide** The instructor should be driving for most of this session. Students were emailed with instructions on how to do the following setup tasks, but walk them through it if they haven't yet completed it or are having trouble. If students run into issues, ask them to share their screens so you can see their issue and other students can learn from the solution.

## Working Process

1. Guide students through the process of finding the email with the dbt Cloud invite, accepting the invite, and finding their project.
2. Have them enter their Snowflake credentials and initialize their project.
3. Commit to main with a commit message like "initialized project".
4. Have them create a new branch so they can start making changes.

Now that the setup is done, do the following.

1. Give them a brief tour of the IDE: the file tree, file editor, the "Preview Data" and "Compile SQL" buttons, the results tabs (Query Results, Compiled SQL, Lineage), and command line.
2. Students will have the following version of `customers.sql` already in their projects. Code snippets are included below in case there are any issues.

### `stg_customers.sql` (original)

```
with customers as (  
    select  
        id as customer_id,  
        first_name,  
        last_name  
    from raw.jaffle_shop.customers  
)  
,  
orders as (  
    select  
        id as order_id,  
        user_id as customer_id,  
        order_date,  
        status  
    from raw.jaffle_shop.orders  
)  
,  
customer_orders as (  
    select  
        customer_id,  
        order_id,  
        order_date,  
        status  
    from orders  
    where user_id is not null  
)
```

```

select
    customer_id,

    min(order_date) as first_order_date,
    max(order_date) as most_recent_order_date,
    count(order_id) as number_of_orders

from orders

group by 1

),

final as (

    select
        customers.customer_id,
        customers.first_name,
        customers.last_name,
        customer_orders.first_order_date,
        customer_orders.most_recent_order_date,
        coalesce(customer_orders.number_of_orders, 0) as number_of_orders

    from customers

    left join customer_orders using (customer_id)

)

select * from final

```

3. Walk students through refactoring `customers.sql` to break out the staging CTEs into the staging models below. Create a subdirectory called staging under the models folder. Replace the CTEs in `customers.sql` with `ref()` functions.

#### staging/stg\_customers.sql

```

select
    id as customer_id,
    first_name,
    last_name

from raw.jaffle_shop.customers

```

#### staging/stg\_orders.sql

```

select
    id as order_id,

```

```
    user_id as customer_id,  
    order_date,  
    status  
  
from raw.jaffle_shop.orders
```

### customers.sql

```
with customers as (  
    select * from {{ ref('stg_customers') }}  
)  
,  
orders as (  
    select * from {{ ref('stg_orders') }}  
)  
,  
customer_orders as (  
    select  
        customer_id,  
  
        min(order_date) as first_order_date,  
        max(order_date) as most_recent_order_date,  
        count(order_id) as number_of_orders  
  
    from orders  
  
    group by 1  
)  
,  
final as (  
    select  
        customers.customer_id,  
        customers.first_name,  
        customers.last_name,  
        customer_orders.first_order_date,  
        customer_orders.most_recent_order_date,  
        coalesce(customer_orders.number_of_orders, 0) as number_of_orders  
  
    from customers  
  
    left join customer_orders using (customer_id)  
  
    )  
  
select * from final
```

4. Change the models block in the `dbt_project.yml` to match the snippet below.

### `dbt_project.yml`

```
# replace only the models block with the code below
models:
  jaffle_shop:
    +materialized: table
  staging:
    +materialized: view
```

## Working Group #2

---

**Facilitation Guide** The instructor should be driving for most of this session. Be sure to allow people time to catch up after each step outlined below.

**Working Process** - Facilitate discussion to name the following steps for building the orders model and refactoring the customers model

1. Inspect `raw.stripe.payment`.
2. Stage payment data as `stg_payments`.
3. Inspect `stg_payments` and `stg_orders`, recognize that orders to payments is one-to-many.
4. Write the `orders` model.
5. Refactor the `customers` model.

### `stg_payments.sql`

```
select
  id as payment_id,
  orderid as order_id,
  paymentmethod as payment_method,
  status,

  -- amount is stored in cents, convert it to dollars
  amount / 100 as amount,
  created as created_at

from raw.stripe.payment
```

### `orders.sql`

```
with orders as (
  select * from {{ ref('stg_orders') }}
),
```

```
payments as (  
    select * from {{ ref('stg_payments') }}  
) ,  
  
order_payments as (  
    select  
        order_id,  
        sum(case when status = 'success' then amount end) as amount  
  
    from payments  
    group by 1  
) ,  
  
final as (  
  
    select  
        orders.order_id,  
        orders.customer_id,  
        orders.order_date,  
        coalesce(order_payments.amount, 0) as amount  
  
    from orders  
    left join order_payments using (order_id)  
)  
  
select * from final
```

### customers.sql (refactored)

```
with customers as (  
    select * from {{ ref('stg_customers') }}  
) ,  
orders as (  
    select * from {{ ref('orders') }}  
) ,  
customer_orders as (  
    select  
        customer_id,  
        min(order_date) as first_order_date,  
        max(order_date) as most_recent_order_date,  
        count(order_id) as number_of_orders,  
        sum(amount) as lifetime_value  
    from orders  
    group by 1  
) ,  
final as (  
    select  
        customers.customer_id,  
        customers.first_name,  
        customers.last_name,  
        customer_orders.first_order_date,
```

```
        customer_orders.most_recent_order_date,  
        coalesce(customer_orders.number_of_orders, 0) as number_of_orders,  
        customer_orders.lifetime_value  
    from customers  
    left join customer_orders using (customer_id)  
)  
select * from final
```

## Working Group #3

---

**Facilitation Guide** Ask people how they want to work by sending you a private message in chat: (1) Independently then check in towards the end (2) Guided with a screen share

### Working Process

1. Add Tests
2. Add Sources and Refactor staging models
3. Add Docs
4. Refactor project into marts/core and staging/jaffle\_shop, staging/stripe

### src\_jaffle\_shop.yml

```
version: 2  
  
sources:  
  - name: jaffle_shop  
    description: A clone of a Postgres application database.  
    database: raw  
    tables:  
      - name: customers  
        description: Raw customers data.  
        columns:  
          - name: id  
            description: Primary key for customers  
            tests:  
              - unique  
              - not_null  
  
      - name: orders  
        description: Raw orders data.  
        columns:  
          - name: id  
            description: Primary key for orders.  
            tests:  
              - unique  
              - not_null
```

### stg\_jaffle\_shop.yml

```
version: 2

models:
- name: stg_customers
  description: Staged customer data from our jaffle shop app.
  columns:
    - name: customer_id
      description: The primary key for customers.
      tests:
        - unique
        - not_null

- name: stg_orders
  description: Staged order data from our jaffle shop app.
  columns:
    - name: order_id
      description: Primary key for orders.
      tests:
        - relationships:
            to: ref('stg_customers')
            field: customer_id
    - name: status
      description: '{{ doc("order_status") }}'
      tests:
        - accepted_values:
            values:
              - completed
              - shipped
              - returned
              - placed
              - return_pending
```

### core.yml

```
version: 2

models:
- name: customers
  columns:
    - name: customer_id
      tests:
        - unique
        - not_null

- name: orders
  description: One record per order
  columns:
    - name: order_id
      tests:
        - unique
        - not_null
```

```

- name: status
  description: "{{ doc('order_status') }}"
  tests:
    - accepted_values:
        values:
            ["placed", "shipped", "completed", "return_pending",
"returned"]

- name: amount
  description: Amount in USD

```

## Working Group #4

---

### Jinja Working Exercise Steps

1. Write the pivot in pure SQL.
2. Write the pivot with some Jinja + SQL (don't address changing payment methods or how to deal with the final column).
3. Address the trailing comma and set in Jinja.
4. Use dbt\_utils to get column values.

**Facilitation Guide** The instructor for the Jinja session will get the class started on the first two steps. Then in breakout rooms, instructors will nominate one students to be the driver for refactoring this query.

### Step 1: Pure SQL

```

with payments as (
    select * from {{ ref('stg_payments') }}
),

pivoted as (
    select
        order_id,

        sum(case when payment_method = 'coupon' then amount else 0 end) as
coupon_amount,
        sum(case when payment_method = 'credit_card' then amount else 0
end) as credit_card_amount,
        sum(case when payment_method = 'bank_transfer' then amount else 0
end) as bank_transfer_amount,
        sum(case when payment_method = 'gift_card' then amount else 0 end)
as gift_card_amount,

        sum(amount) as total

    from payments

    group by 1

)

```



## Step 2: Some Jinja and SQL

```
-- can we use {% set %} for our payment method
-- what happens if there's a new payment method
-- can we make a macro?
with payments as (
    select * from {{ ref('stg_payments') }}
),

pivoted as (
    select
        order_id,

        {% for payment_method in ['credit_card', 'coupon',
'bank_transfer', 'gift_card'] %}

            sum(case when payment_method = '{{ payment_method }}' then amount
else 0 end) as {{ payment_method }}_amount,
            -- how to handle trailing comma? (if we remove the last column)

        {% endfor %}
    from payments

    group by 1
)
```

## Step 3: Address the trailing comma and set in Jinja

- Use the jinja docs to handle the trailing column with if loop.last
- Use set at the top of the model

```
-- can we use {% set %} for our payment method
-- what happens if there's a new payment method
-- can we make a macro?

{% set payment_methods = ['credit_card', 'coupon', 'bank_transfer',
'gift_card'] %}

with payments as (
    select * from {{ ref('stg_payments') }}
),

pivoted as (
    select
        order_id,

        {% for payment_method in payment_methods %}
```

```

        sum(case when payment_method = '{{ payment_method }}' then amount
else 0 end) as {{ payment_method }}_amount

        {% if not loop.last %}
        ,
        {% endif %}

    {% endfor %}
from payments

group by 1

)

```

#### Step 4: Get column values with macro

- Import dbt\_utils
- change the set to be the `get_column_values` macro

#### packages.yml

```

packages:
- package: dbt-labs/dbt_utils
  version: 0.8.0

```

#### payments\_\_pivoted.sql

```

{% set payment_methods =
dbt_utils.get_column_values(table=ref('stg_payments'),
column='payment_method') -%}

with payments as (
    select * from {{ ref('stg_payments') }}
),

pivoted as (
    select
        order_id,

        {%- for payment_method in payment_methods -%}

        sum(case when payment_method = '{{ payment_method }}' then amount
else 0 end) as {{ payment_method }}_amount

        {%- if not loop.last -%}
        ,
        {% endif -%}

```

```
        {%– endfor %}

    from payments
    group by 1
)

select * from pivoted
```

## Working Group #5 (optional)

---

This is meant to be a *capstone* of sorts for learners to show what they know! The training wheels are not completely off yet, so use this guide to provide scaffolding for learners:

**Facilitation Guide** Ask people how they want to work by sending you a private message in chat: (1) Independently then check in towards the end (2) Guided with a screen share

**Working Process** - Facilitate discussion to name the following steps for creating a final model. There is no **correct** process here, but learners should be comfortable with the idea of refactoring

1. Create a source for the three ticket tailor tables
2. Create a staging model for each raw table (pro tip: use the codegen package)
3. Create a fct\_tickets model for answering the question.
4. Refactor, refactor, refactor.
5. Formalize with tests and documentation

### Code Snippets to assist:

#### stg\_tt\_orders

```
with source as (

    select * from {{ source('ticket_tailor', 'orders') }}

),

renamed as (

    select
        -- keys
        id as order_id,
        txn_id as transaction_id,

        -- descriptions
        object,
        currency,
        round(subtotal / 100, 2) as subtotal,
        round(tax / 100) as tax,
```

```

        round(total / 100) as total,
        round(refund_amount / 100) as refund_amount,
        event_summary,
        line_items,

        -- status
        status,

        -- timestamps
        to_timestamp_ntz(created_at) as created_at,

        -- metadata
        _sdc_batched_at,
        _sdc_received_at,
        _sdc_sequence,
        _sdc_table_version

    from source

)

select * from renamed

```

### stg\_tt\_events

```

with source as (

    select * from {{ source('ticket_tailor', 'events') }}

),

renamed as (

    select
        -- keys
        id as event_id,

        -- descriptions
        name,
        description,
        object,
        payment_methods,
        images,
        ticket_types,
        currency,
        timezone,
        url,
        venue,
        call_to_action,

        -- status
        status,

```

```

        total_issued_tickets,
        total_orders,

        -- booleans
        online_event as is_online_event,
        private as is_private,
        tickets_available as is_tickets_available,

        -- timestamps
        to_timestamp_ntz(created_at) as created_at,

        -- metadata
        _sdc_batched_at,
        _sdc_received_at,
        _sdc_sequence,
        _sdc_table_version

        -- ignored
        -- 'end',
        -- 'start'
        -- ticket_groups

    from source

)

select * from renamed

```

### stg\_tt\_tickets

```

with source as (

    select * from {{ source('ticket_tailor', 'issued_tickets') }}

),

renamed as (

    select
        -- keys
        id as ticket_id,
        ticket_type_id,
        event_id,
        order_id,

        -- descriptions
        object,
        barcode,
        barcode_url,

        -- status
        status,

```

```
-- timestamps
to_timestamp_ntz(created_at) as created_at,
to_timestamp_ntz(updated_at) as updated_at,
to_timestamp_ntz(voided_at) as voided_at,

-- metadata
_sdc_batched_at,
_sdc_received_at,
_sdc_sequence,
_sdc_table_version

-- ignored

from source

)

select * from renamed
```

### fct\_tickets

```
with tickets as (
    select * from {{ ref('stg_tt_tickets') }}
),
events as (
    select * from {{ ref('stg_tt_events') }}
),
orders as (
    select * from {{ ref('stg_tt_orders') }}
),
joined as (
    select
        ticket_id,
        event_id,
        order_id,
        created_at,
        updated_at,
        events.name as event_name,
        events.timezone as event_timezone,
        orders.total as ticket_amount,
        case
            when orders.status = 'completed' then false
            when orders.status = 'cancelled' then true
        end as is_refunded
    from tickets
    left join events using (event_id)
    left join orders using (order_id)
)
```

```
select * from joined
```