Solving the N + 1 Query problem

For Microservices Without any framework

Agenda

- What is the N + 1 query problem?
- Why it is harder to solve in a microservice architecture?
- How to build a user-friendly API to solve the problem?

Assumptions

- The talk is really about building an API using Java 8+ functional patterns
- Not about databases, JPA or Hibernate
- Mostly not about high level microservice architectures
- Very code oriented focusing on core java

What is the N + 1 query problem?

Example: *Online Store*

Get a list of all items purchased by some customers

```
Select * from customer where…
```

What is the N + 1 query problem?

Assuming we have 10 customers, for each 10 customers c :

```
Select * from order_item i
   where i.customer_id = c.customer_id
```

What is the N + 1 query problem?

How many queries issued to the database?

What is the N + 1 query problem?

Answer: 11 (10 + 1)

- 1 query to `customer`
- 10 queries to `order_item`

What is the N + 1 query problem?

This is linear… until… it’s not anymore!

- 100 customers… 101 queries
- 500 customers… 501 queries
- 1000 customers… 0 query ;-)

So what should happen instead?

- 10, 100, 500 customers… doesn’t matter
- Only 1 query to `order_item`
- Join each java Customer entity with each associated OrderItem

ORMs to the rescue?

- JPA query with Hibernate for example (pseudo-code):

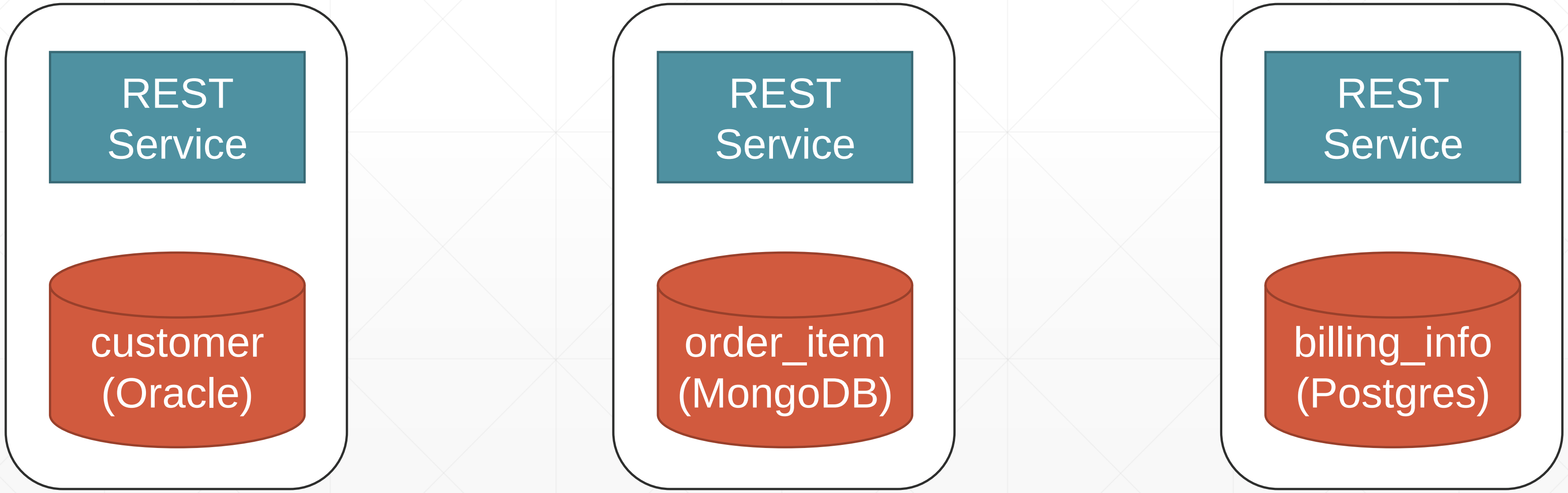
```
Select c from Customer c
   JOIN FETCH c.orderItems
```

Will execute behind the scene a query looking like:

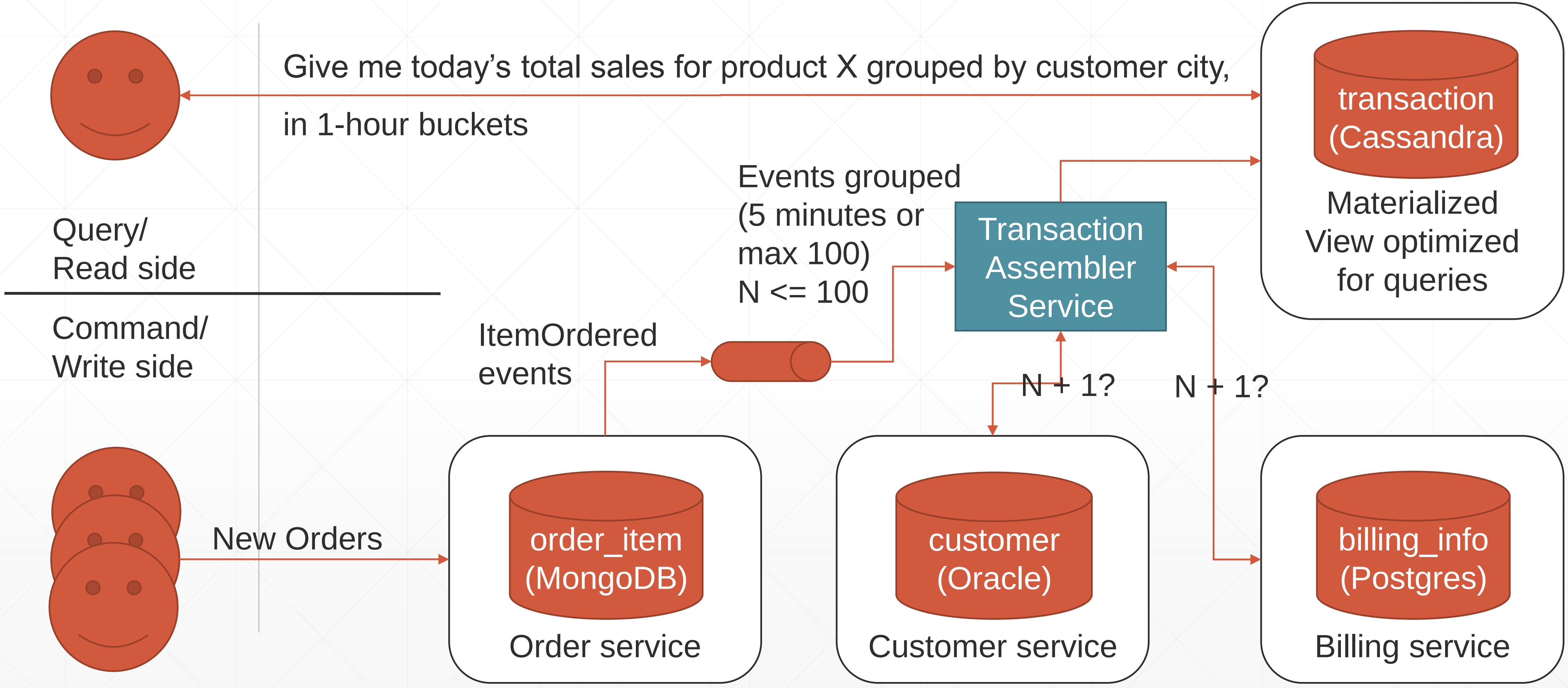
```
select … from customer c0
   inner join order_item o1
   on c0.customer_id = o1.customer_id
```

But in a Microservice Environment…

- ORMs works with a single database
- Now we need to aggregate disconnected sources of data…

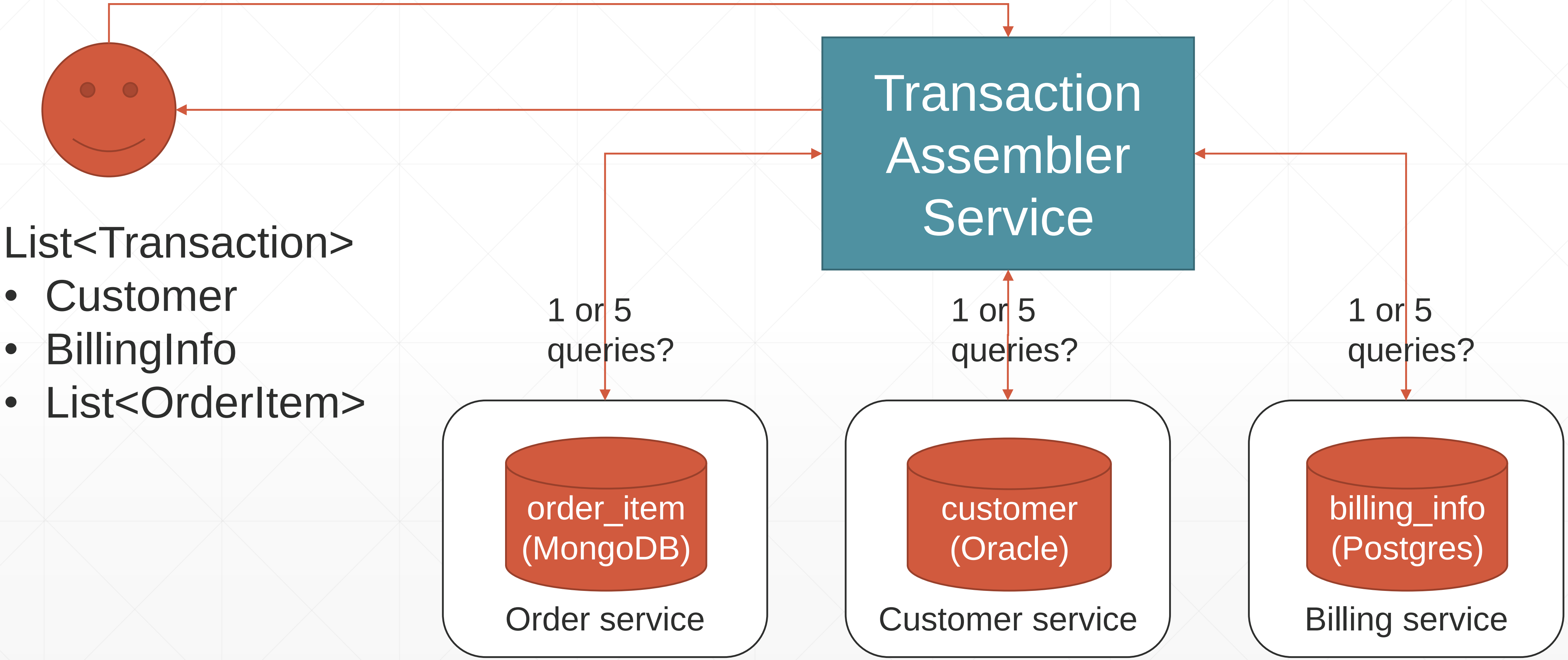


Fictional ES/CQRS Microservice Scenario



A Simpler Example

Give me all transactions for customer id 1, 2, 3, 4, 5

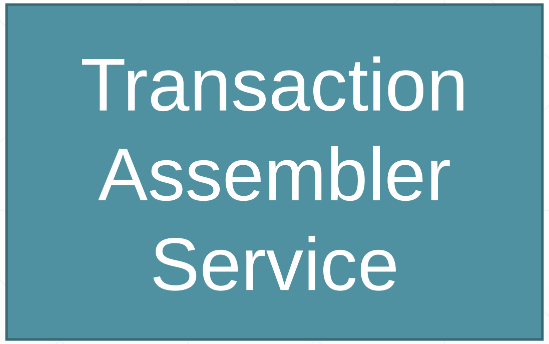


List<Transaction>

- Customer
- BillingInfo
- List<OrderItem>

So how do we implement that Box?

- Without triggering the N + 1 Query Problem


Transaction Assembler Service

Assuming...

```java
@Data
@AllArgsConstructor
public class Customer {
    private final Long customerId;
    private final String name;
}
```

And...

```
@Data
@AllArgsConstructor
public class BillingInfo {
    private final Long billingInfoId
    private final Long customerId;
    private final String creditCardNumber;
}
```

And…

```
@Data
@AllArgsConstructor
public class OrderItem {
    private final Long orderItemId
    private final Long customerId;
    private final String orderDescription;
    private final Double price;
}
```

And Our Aggregate…

```java
@Data
@AllArgsConstructor
public class Transaction {
    private final Customer customer;
    private final BillingInfo billingInfo;
    private final List<OrderItem> orderItems;
}
```

Now our disparate data sources…

Spring Data JPA Repository on Oracle Database

```
@Query("select c from Customer c where c.customerId in :ids")
List<Customer> getCustomers(List<Long> customerIds);
```

And…

REST HTTP or Binary Protocol Client to BillingInfo Service

```
List<BillingInfo> getBillingInfo(List<Long> customerIds);
```

Notice the method signature?

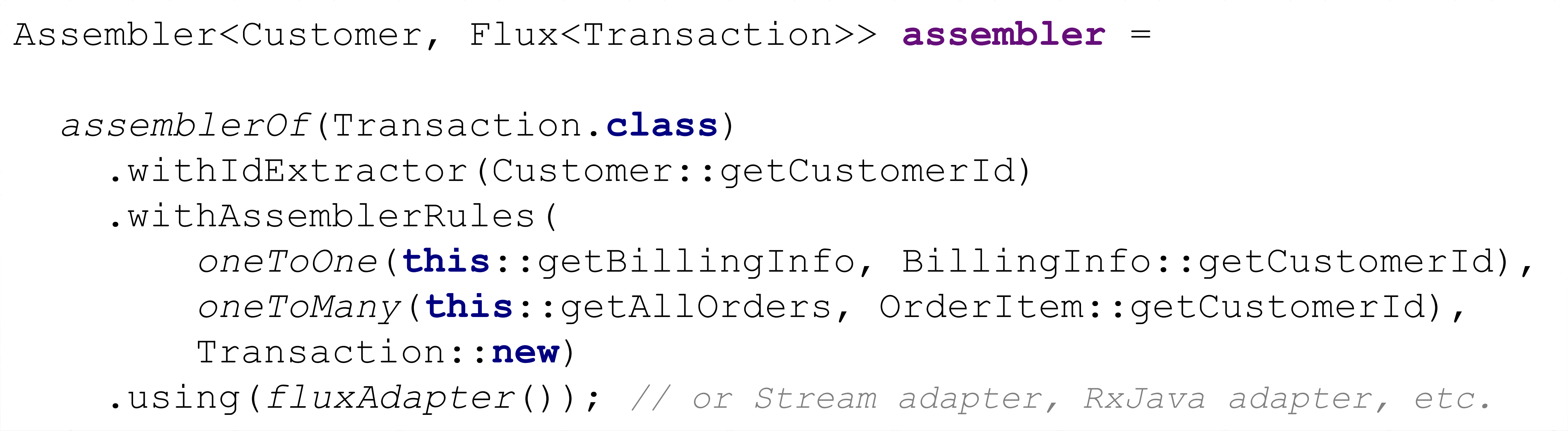
And…

Spring Data MongoDB Repository

```
@Query("{ 'customerId' : { $in: ?0 } }")
List<OrderItem> getAllOrders(List<Long> customerIds);
```

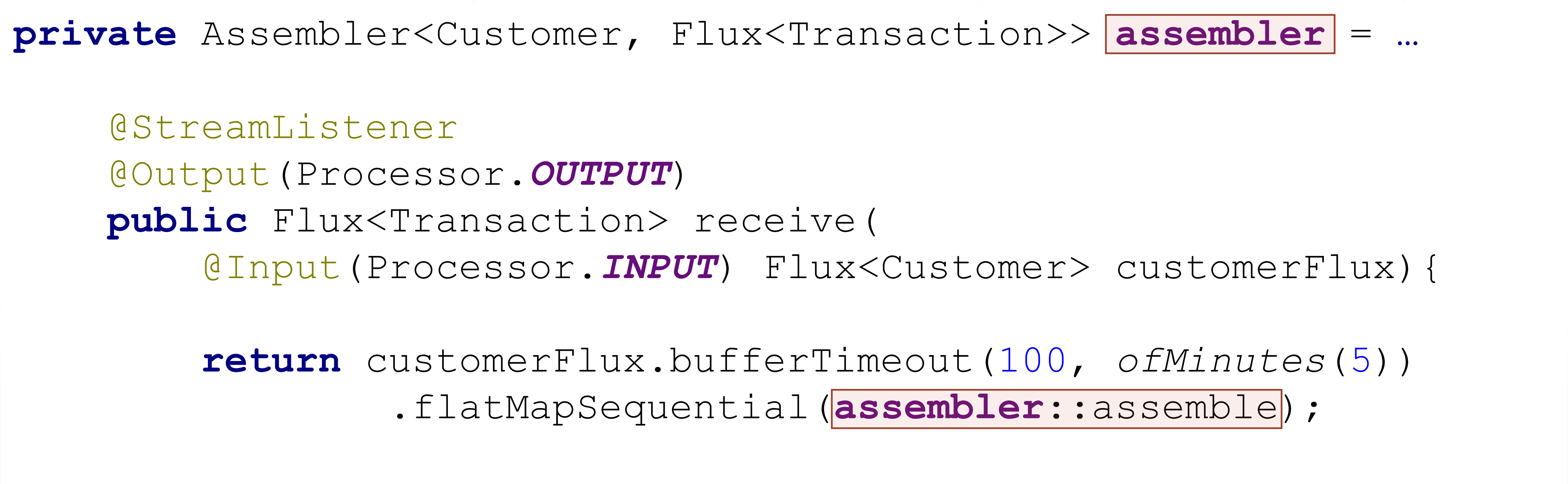
Notice again the method signature?

This is the API we want to build



```java
Assembler<Customer, Flux<Transaction>> assembler =

  assemblerOf(Transaction.class)
    .withIdExtractor(Customer::getCustomerId)
    .withAssemblerRules(
        oneToOne(this::getBillingInfo, BillingInfo::getCustomerId),
        oneToMany(this::getAllOrders, OrderItem::getCustomerId),
        Transaction::new)
    .using(fluxAdapter()); // or Stream adapter, RxJava adapter, etc.
```

Usage example with Spring Cloud Stream (Kafka) + Project Reactor



```
private Assembler<Customer, Flux<Transaction>> assembler = …

    @StreamListener
    @Output(Processor.OUTPUT)
    public Flux<Transaction> receive(
        @Input(Processor.INPUT) Flux<Customer> customerFlux){

        return customerFlux.bufferTimeout(100, ofMinutes(5))
                .flatMapSequential(assembler::assemble);
```

How do we build such an API?

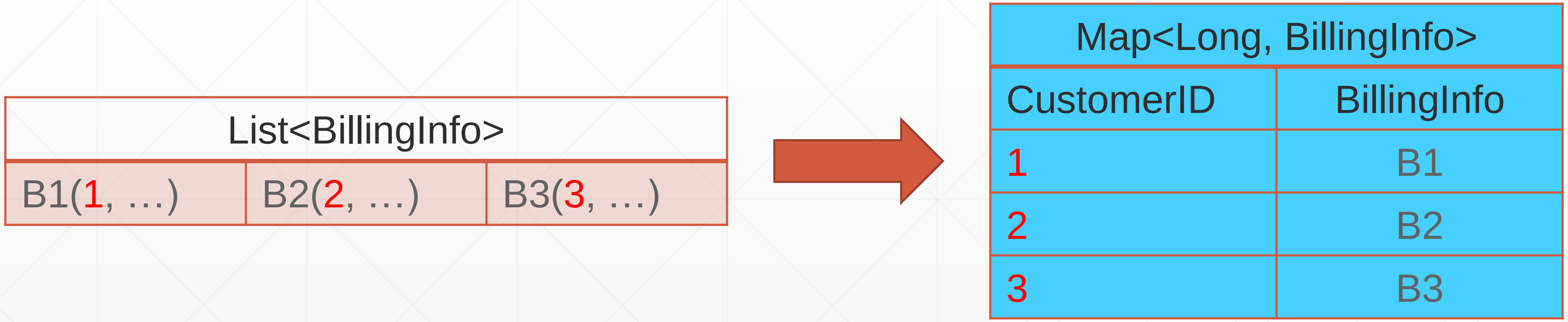
1. Implement oneToOne and oneToMany semantics for sub-queries
2. Join/Aggregation logic
3. Decouple Join logic from execution engines
   - Java 8 Stream
   - Project Reactor
   - RxJava
   - Akka Stream

1. Implement oneToOne and oneToMany semantics

```
assemblerOf(Transaction.class)
    .withIdExtractor(Customer::getCustomerId)
    .withAssemblerRules(
        oneToOne(this::getBillingInfo, BillingInfo::getCustomerId),
        oneToMany(this::getAllOrders, OrderItem::getCustomerId),
        Transaction::new)
    .using(fluxAdapter());
```

Let’s start with a simple utility method…

```
Map<Long, BillingInfo> billingInfoMap =
    queryOneToOne(of(1L, 2L, 3L),
        this::getBillingInfo, // REST HTTP call
        BillingInfo::getCustomerId);
```

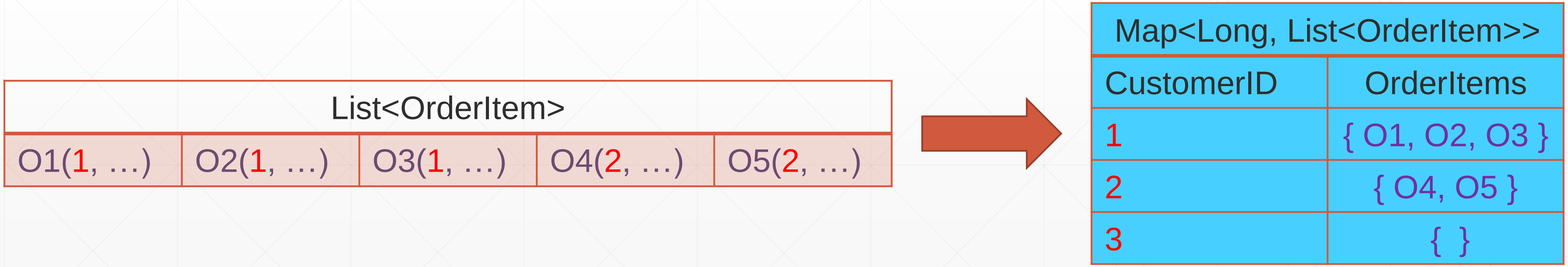


List<BillingInfo>		
B1(1, …)	B2(2, …)	B3(3, …)

Map<Long, BillingInfo>	
CustomerID	BillingInfo
1	B1
2	B2
3	B3

Let’s start with a simple utility method…

```
Map<Long, List<OrderItem>> orderItemMap =
    queryOneToMany(of(1L, 2L, 3L),
        this::getAllOrders, // Spring Data MongoDB
        OrderItem::getCustomerId,
        ArrayList::new);
```



List<OrderItem>				
O1(1, …)	O2(1, …)	O3(1, …)	O4(2, …)	O5(2, …)

Map<Long, List<OrderItem>>	
CustomerID	OrderItems
1	{ O1, O2, O3 }
2	{ O4, O5 }
3	{ }

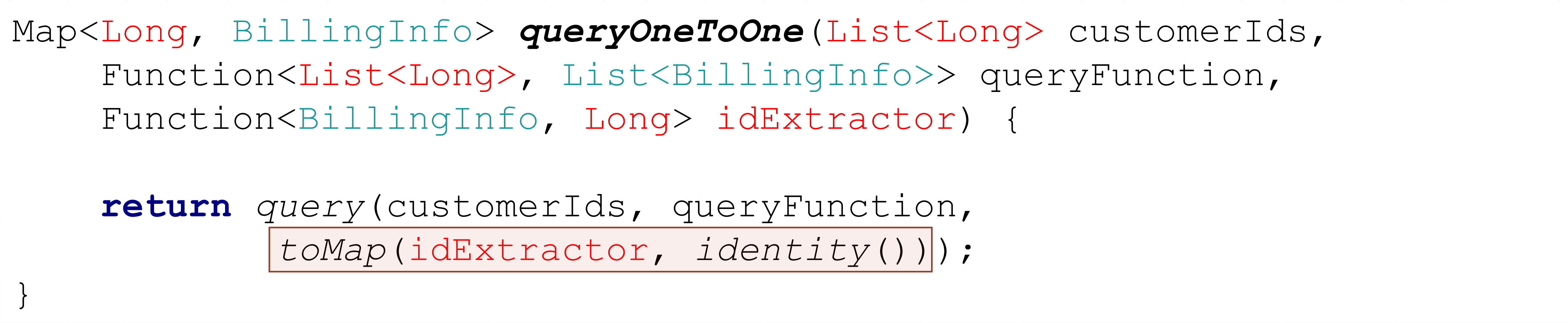
queryOneToOne vs. queryOneToMany

- Both queryOneToOne and queryOneToMany are very similar
- They both retrieve list of data for a data source
- They both group the data received (by ID)
- They only differ in how they group the data

Strategy pattern with Java 8 Stream collectors

```java
static <ID, R, IDC extends Collection<ID>, RC extends
Collection<R>>

Map<ID, R> queryOneToOne(IDC ids,
              Function<IDC, RC> queryFunction,
              Function<R, ID> idExtractor) {

    return query(ids, queryFunction,
            toMap(idExtractor, identity()));
}
```

Strategy pattern with Java 8 Stream collectors

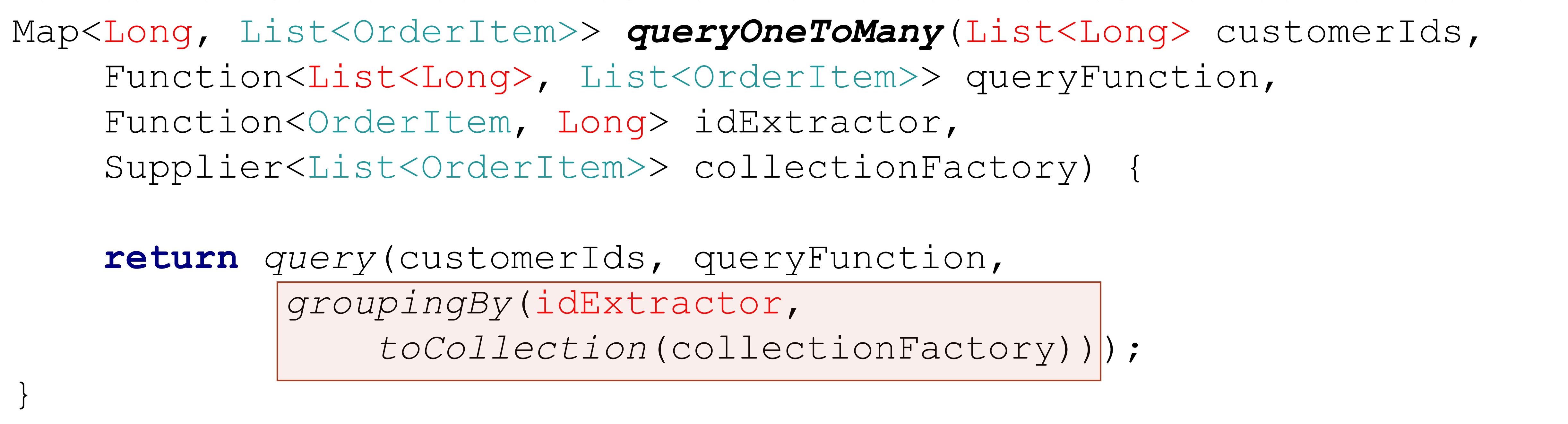


```
Map<Long, BillingInfo> queryOneToOne(List<Long> customerIds,
    Function<List<Long>, List<BillingInfo>> queryFunction,
    Function<BillingInfo, Long> idExtractor) {

    return query(customerIds, queryFunction,
                toMap(idExtractor, identity()));
}
```

Strategy pattern with Java 8 Stream collectors

```
Map<ID, RC> queryOneToMany(IDC ids,
               Function<IDC, RC> queryFunction,
               Function<R, ID> idExtractor,
               Supplier<RC> collectionFactory) {

    return query(ids, queryFunction,
            groupingBy(idExtractor,
                toCollection(collectionFactory)));
}
```

Strategy pattern with Java 8 Stream collectors



```
Map<Long, List<OrderItem>> queryOneToMany(List<Long> customerIds,
    Function<List<Long>, List<OrderItem>> queryFunction,
    Function<OrderItem, Long> idExtractor,
    Supplier<List<OrderItem>> collectionFactory) {

    return query(customerIds, queryFunction,
                 groupingBy(idExtractor,
                     toCollection(collectionFactory)));
}
```

Strategy pattern with Java 8 Stream collectors

Our generic query() method:

```
Map<ID, V> query(IDC ids,
                 Function<IDC, RC> queryFunction,
                 Collector<R, ?, Map<ID, V>> mapCollector) {

     return queryFunction.apply(ids).stream()
                 .collect(mapCollector);
}
```

V = R || RC (e.g. List<R>)

From queryOneToXXX to oneToXXX

Function returning function: From eager to lazy execution

```
Mapper<ID, R> oneToOne(
        Function<IDC, RC> queryFunction,
        Function<R, ID> idExtractor) {

    return entityIds -> queryOneToOne(
                 (IDC) entityIds, queryFunction, idExtractor);
}
```

Function returning function:
From eager to lazy execution

```
@FunctionalInterface
public interface Mapper<ID, R> {

    Map<ID, R> apply(Iterable<ID> entityIds);
}
```

- Like `java.util.Function` with predefined types
- Equivalent to `Function<Iterable<ID>, Map<ID, R>>`

Function returning function: From eager to lazy execution

```
Mapper<Long, BillingInfo> billingInfoMapper =
    oneToOne(this::getBillingInfo, BillingInfo::getCustomerId);
```

- Move `billingInfoMapper` around, pass it to other methods…
- Later invoke it…

```
Map<Long, BillingInfo> map =
        billingInfoMapper.apply(of(1L, 2L, 3L));
```

Function returning function:
From eager to lazy execution

You can guess the implementation of `oneToMany`…

Function returning function: From eager to lazy execution

```
Mapper<ID, RC> oneToMany(
        Function<IDC, RC> queryFunction,
        Function<R, ID> idExtractor,
        Supplier<RC> collectionFactory) {

    return entityIds -> queryOneToMany(
                (IDC) entityIds, queryFunction, idExtractor,
                collectionFactory);
}
```

2. Implement Join/Aggregate Logic (part 1)

```
assemblerOf(Transaction.class)
    .withIdExtractor(Customer::getCustomerId)
    .withAssemblerRules(
        oneToOne(this::getBillingInfo, BillingInfo::getCustomerId),
        oneToMany(this::getAllOrders, OrderItem::getCustomerId),
        Transaction::new)
    .using(fluxAdapter());
```

We could have 3, 5, 10 overloaded methods…

```
withAssemblerRules(
    Mapper<ID, E1> mapper,
    BiFunction<T, E1, R> assemblerFunction)
```

```
withAssemblerRules(
    Mapper<ID, E1> mapper1,
    Mapper<ID, E2> mapper2,
    Function3<T, E1, E2, R> assemblerFunction)
```

But wouldn’t it be so easier to just use…

```
withAssemblerRules(
    List<Mapper<ID, ?>> mappers,
    BiFunction<T, Object[], R> aggregationFunction);
```

… and reuse the same code for an arbitrary number of parameters instead?

Sure, except for the fact that we are losing type information…

We want the compiler to catch this…

```
withAssemblerRules(
  List.of(
    oneToMany(this::getAllOrders, OrderItem::getCustomerId),
    oneToOne(this::getBillingInfo, BillingInfo::getCustomerId)
  ),
  (Customer c, Object[] subQueryResults) ->
      new Transaction(c, (BillingInfo)subQueryResults[0],
                         (List<OrderItem>)subQueryResults[1])
)
```

This smells ClassCastException using a generic solution…

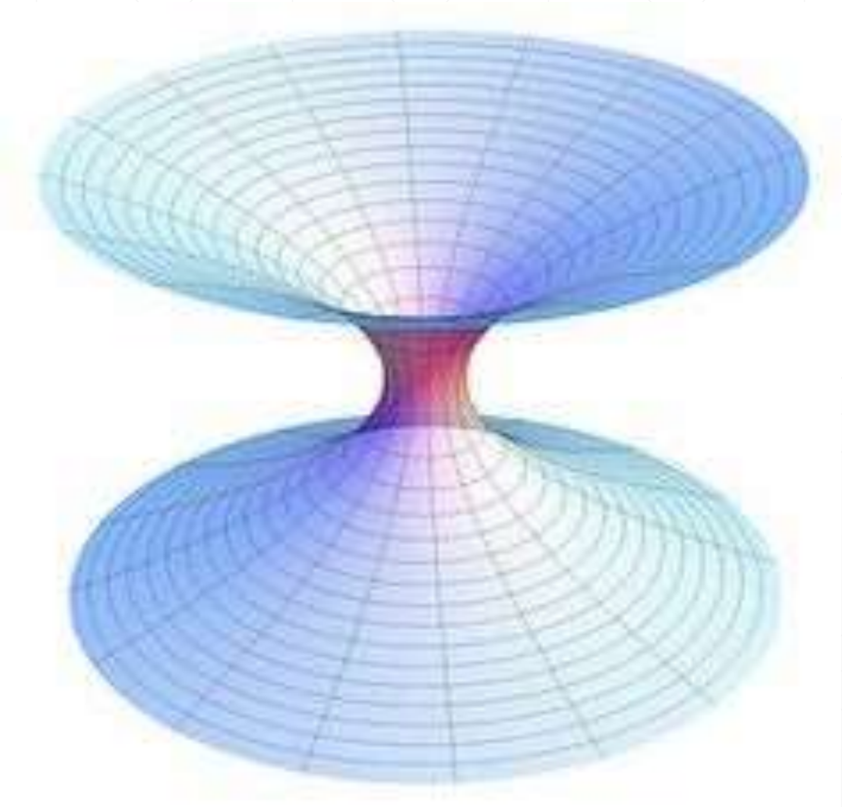
So how to get from…

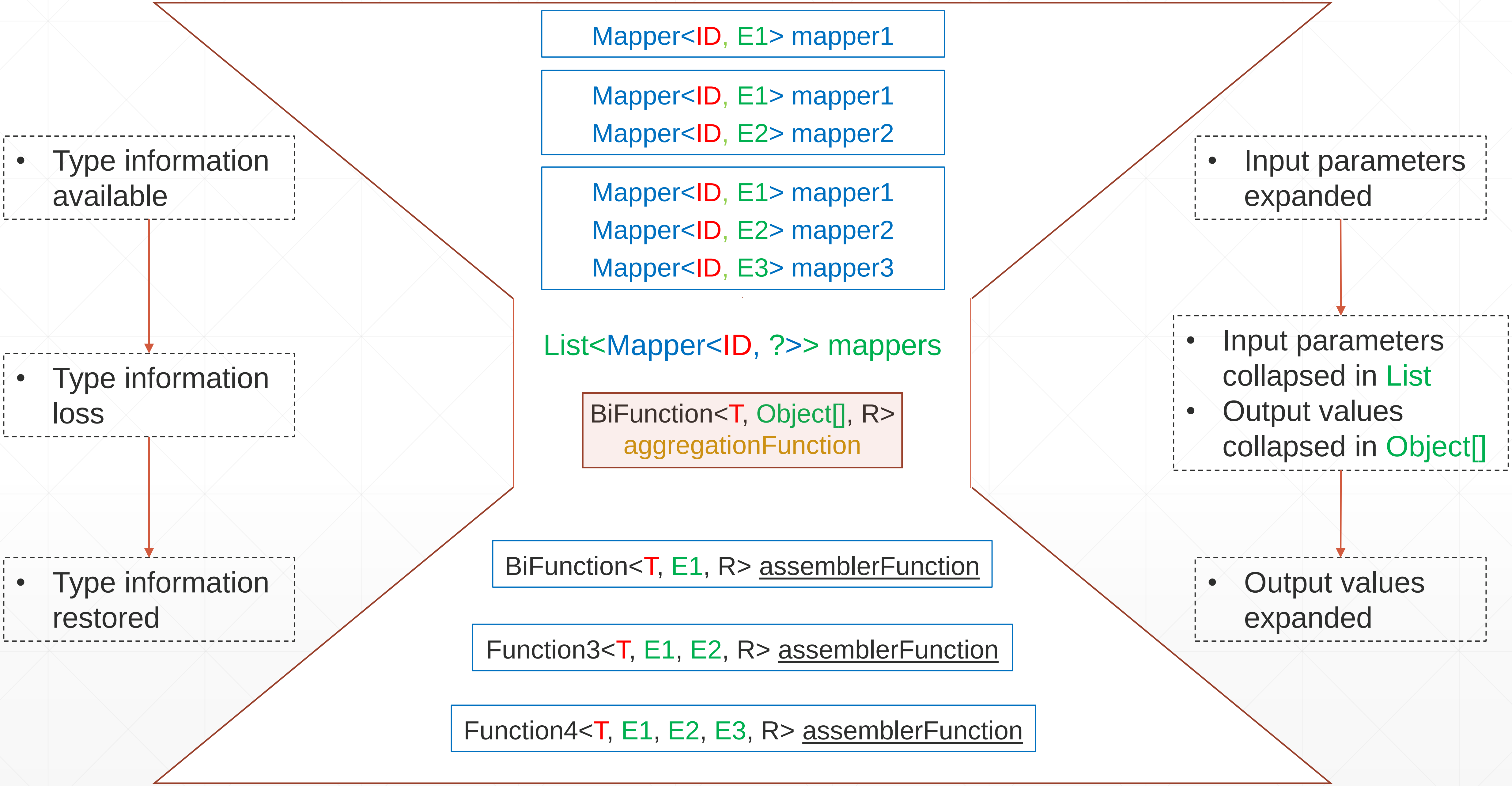
```
BiFunction<T, Object[], R> aggregationFunction
```

Back to…

```
BiFunction<T, E1, R> assemblerFunction)

Function3<T, E1, E2, R> assemblerFunction)
```

Double Sided Funnel Pattern
(You won’t find it on the Internet ☺)





- Type information available

- Type information loss

- Type information restored

Mapper<ID, E1> mapper1

Mapper<ID, E1> mapper1
Mapper<ID, E2> mapper2

Mapper<ID, E1> mapper1
Mapper<ID, E2> mapper2
Mapper<ID, E3> mapper3

List<Mapper<ID, ?>> mappers

BiFunction<T, Object[], R>
aggregationFunction

BiFunction<T, E1, R> assemblerFunction

Function3<T, E1, E2, R> assemblerFunction

Function4<T, E1, E2, E3, R> assemblerFunction

- Input parameters expanded

- Input parameters collapsed in List
- Output values collapsed in Object[]

- Output values expanded

Assuming this…

```
Function3<T, E1, E2, R> assemblerFunction =
```

```
(Customer c, BillingInfo b, List<OrderItem> o) -> new Transaction(c, b, o))
```

… with our generic method called by our 2, 5, 10 overloaded methods of `withAssemblerRules`

```
withAssemblerRules(
    List<Mapper<ID, ?>> mappers,
    BiFunction<T, Object[], R> aggregationFunction);
```

Double Sided Funnel Pattern with 2 Parameters

```
<E1, E2> AssembleUsingBuilder<T, ID, R> withAssemblerRules(
        Mapper<ID, E1> mapper1,
        Mapper<ID, E2> mapper2,
        Function3<T, E1, E2, R> assemblerFunction) {

    return withAssemblerRules(List.of(mapper1, mapper2),
            (topLevelEntity, mapperResults) ->
                    assemblerFunction.apply(topLevelEntity,
                                                (E1) mapperResults[0],
                                                (E2) mapperResults[1]));
}
```

Double Sided Funnel Pattern with 3 Parameters

```
<E1, E2, E3> AssembleUsingBuilder<T, ID, R> withAssemblerRules(
        Mapper<ID, E1> mapper1,
        Mapper<ID, E2> mapper2,
        Mapper<ID, E3> mapper3,
        Function4<T, E1, E2, E3, R> assemblerFunction) {

    return withAssemblerRules(List.of(mapper1, mapper2, mapper3),
            (topLevelEntity, mapperResults) ->
                assemblerFunction.apply(topLevelEntity,
                                         (E1) mapperResults[0],
                                         (E2) mapperResults[1],
                                         (E3) mapperResults[2]));
}
```

2. Implement Join/Aggregate Logic (part 2)

```
assemblerOf(Transaction.class)
    .withIdExtractor(Customer::getCustomerId)
    .withAssemblerRules(
        oneToOne(this::getBillingInfo, BillingInfo::getCustomerId),
        oneToMany(this::getAllOrders, OrderItem::getCustomerId),
        Transaction::new)
    .using(fluxAdapter());
```

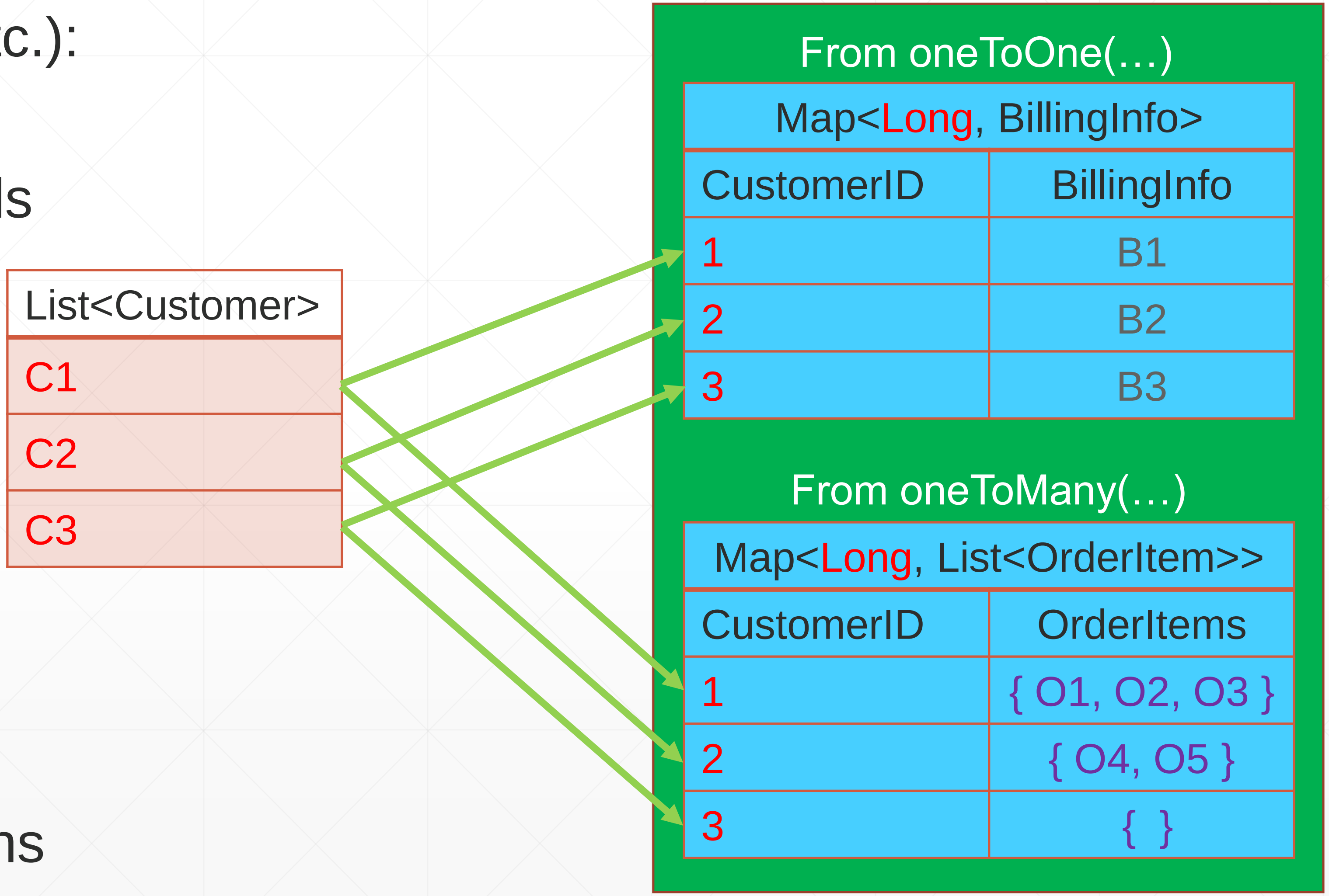
Join/Aggregate function

For 3 customers (or 20, 40, etc.):

- ALWAYS only 2 network calls
  - getBillingInfo
  - getAllOrders

- For each customer
  - For each Map<Long, ?>
    - Map.get(customerId)

- 3 x 2 = 6 in-memory iterations



List<Customer>
C1
C2
C3

From oneToOne(…)

Map<Long, BillingInfo>	
CustomerID	BillingInfo
1	B1
2	B2
3	B3

From oneToMany(…)

Map<Long, List<OrderItem>>	
CustomerID	OrderItems
1	{ O1, O2, O3 }
2	{ O4, O5 }
3	{ }

Join/Aggregate function

- For each customer
  - For each Map<Long, ?>
    - Map.get(customerId)

This is the output when called for each 3 customers:

```
aggregationFunction.apply( C1, [ B1, { O1, O2, O3 } ] )
aggregationFunction.apply( C2, [ B2, { O4, O5 } ] );
aggregationFunction.apply( C3, [ B3, {  } ] );
```

```
BiFunction<T, List<Map<ID, ?>>, R> joinMapperResultsFunction =
    (topLevelEntity, listOfMapperResults) ->
        aggregationFunction.apply(topLevelEntity,
            listOfMapperResults.stream()
                .map(mapperResult ->
                     mapperResult.get(idExtractor.apply(topLevelEntity)))
                .toArray());
```

Join/Aggregate function

So we have our Join/Aggregate logic wrapped in a reusable function…

But nothing is executed yet, so far we are just writing the recipe…

It’s Execution Time!!!

```
RC assemble(C topLevelEntities, // List<Customer>
    Function<T, ID> idExtractor, // Customer::getCustomerId
    List<Mapper<ID, ?>> subQueryMappers,
    BiFunction<T, Object[], R> aggregationFunction,
    AssemblerAdapter<ID, R, RC> assemblerAdapter)
```

Step 1

- Extract customer ids from our list of customers:

```
List<ID> ids = topLevelEntities.stream()
        .map(idExtractor) // Customer::getCustomerId
        .collect(toList());
```

Step 2

- Convert our mappers into `Supplier<Map<ID, ?>>`
- i.e. we transform 1 parameter function into 0 parameter function by capturing entityIDs in closure
  - Like we did with queryOneToXXX earlier

```
Stream<Supplier<Map<ID, ?>>> mapperSourceSuppliers =
    subQueryMappers.stream()
        .map(mapper -> () -> mapper.apply(ids));
```

Step 3

- For each **customer**:
  - Reuse our `joinMapperResultFunction`

```
Function<List<Map<ID, ?>>, Stream<R>> aggregateStreamBuilder =
    mapperResults -> topLevelEntities.stream()
        .map(topLevelEntity ->
            joinMapperResultsFunction.apply(topLevelEntity,
                                            mapperResults));
```

```
<R> = <Transaction>
```

Step 4

- Let pluggable execution engine do the work
  - Java 8 Steams
  - CompletableFuture
  - Project Reactor
  - RxJava
  - Akka Stream

```
return assemblerAdapter.convertMapperSources(
            mapperSourceSuppliers, aggregateStreamBuilder);
```

3. Decoupling Join/Aggregation Logic from Execution Model

```java
assemblerOf(Transaction.class)
    .withIdExtractor(Customer::getCustomerId)
    .withAssemblerRules(
        oneToOne(this::getBillingInfo, BillingInfo::getCustomerId),
        oneToMany(this::getAllOrders, OrderItem::getCustomerId),
        Transaction::new)
    .using(fluxAdapter());
```

Decoupling Join/Aggregate Logic from Execution Model

```
/**
 * @param <ID> e.g. {@code <Long>}
 * @param <R>  e.g. {@code <Transaction>}
 * @param <RC> e.g. {@code Stream<Transaction>} or {@code Flux<Transaction>}
 */
@FunctionalInterface
public interface AssemblerAdapter<ID, R, RC> {

    RC convertMapperSources(
        Stream<Supplier<Map<ID, ?>>> mapperSourceSuppliers,
        Function<List<Map<ID, ?>>, Stream<R>> aggregateStreamBuilder);
}
```

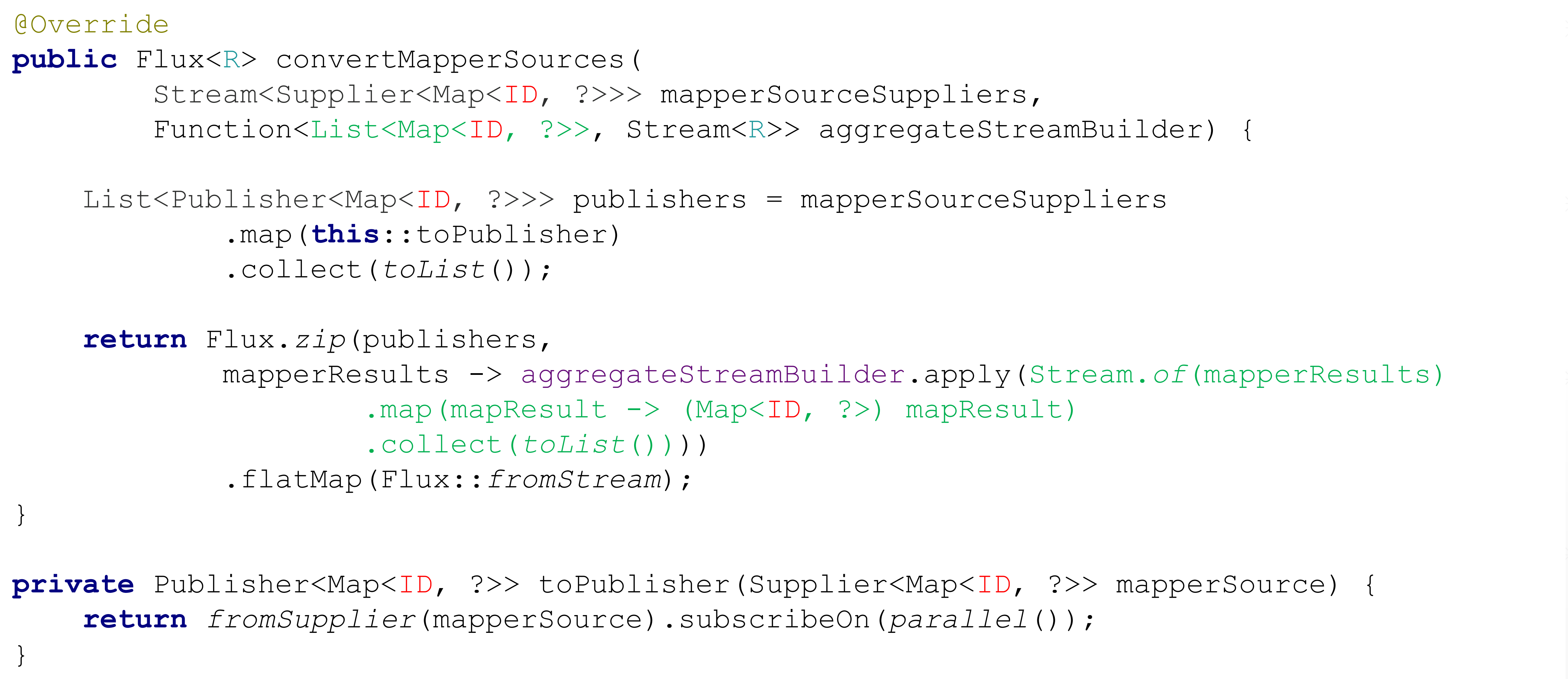
Adapter Responsibilities…

- Convert Supplier<Map<ID, ?>> to adapter specific type
- Trigger the execution of the suppliers
  - Will perform network calls (e.g. getBillingInfo(), getAllOrders())
- Pass the returned List<Map<ID, ?>> to `aggregateStreamBuilder`
- Done!

Java 8 Stream Adapter

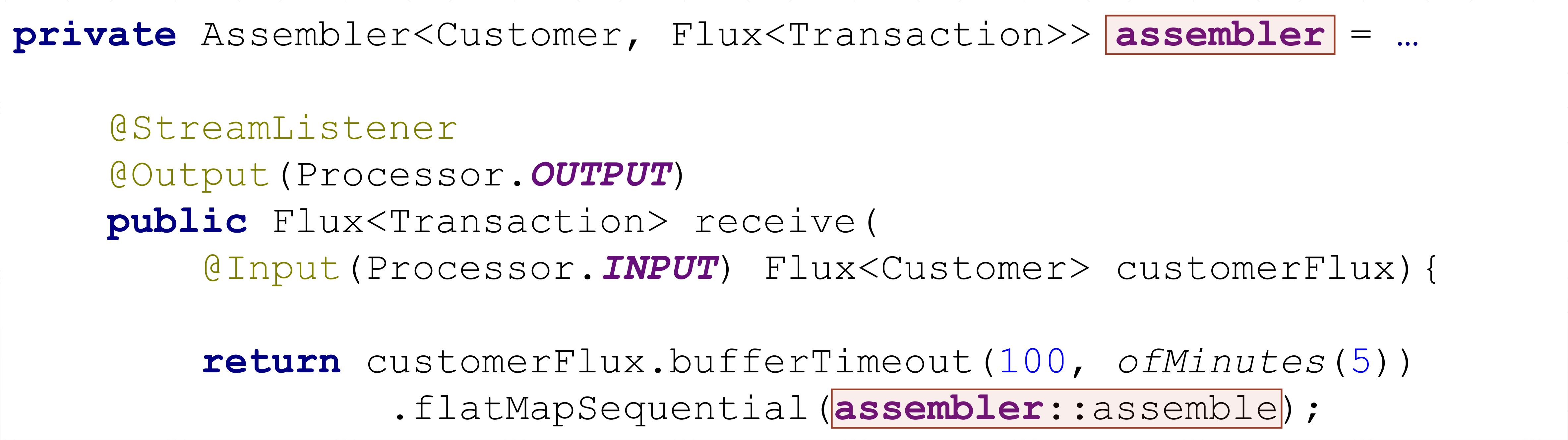
```java
@Override
public Stream<R> convertMapperSources(
        Stream<Supplier<Map<ID, ?>>> mapperSourceSuppliers,
        Function<List<Map<ID, ?>>, Stream<R>> aggregateStreamBuilder) {

    List<Map<ID, ?>> mappers = mapperSourceSuppliers
            .map(Supplier::get)
            .collect(toList());

    return aggregateStreamBuilder.apply(mappers);
}
```

Flux Adapter (Project Reactor)



```
@Override
public Flux<R> convertMapperSources(
        Stream<Supplier<Map<ID, ?>>> mapperSourceSuppliers,
        Function<List<Map<ID, ?>>, Stream<R>> aggregateStreamBuilder) {

    List<Publisher<Map<ID, ?>>> publishers = mapperSourceSuppliers
            .map(this::toPublisher)
            .collect(toList());

    return Flux.zip(publishers,
            mapperResults -> aggregateStreamBuilder.apply(Stream.of(mapperResults)
                    .map(mapResult -> (Map<ID, ?>) mapResult)
                    .collect(toList())))
            .flatMap(Flux::fromStream);
}

private Publisher<Map<ID, ?>> toPublisher(Supplier<Map<ID, ?>> mapperSource) {
    return fromSupplier(mapperSource).subscribeOn(parallel());
}
```

Remember This?



```
private Assembler<Customer, Flux<Transaction>> assembler = …

    @StreamListener
    @Output(Processor.OUTPUT)
    public Flux<Transaction> receive(
        @Input(Processor.INPUT) Flux<Customer> customerFlux){

        return customerFlux.bufferTimeout(100, ofMinutes(5))
                .flatMapSequential(assembler::assemble);
```

Thank You!

http://bit.ly/javatechw