Building upon the info given by @dnickless, I was able to solve this. I'll post the complete solution in the hopes it helps someone else in the future.

I'm using [mongodb-driver:3.6.4](https://mvnrepository.com/artifact/org.mongodb/mongo-java-driver/3.6.4)

First, I had to create a custom aggregation operation class so that I could pass in a custom JSON mongodb query to be used in the aggregation operation. This will allow me to use pipeline within a $lookup which is not supported with the driver version I am using.

public class CustomProjectAggregationOperation implements AggregationOperation {

private String jsonOperation;

public CustomProjectAggregationOperation(String jsonOperation) {

this.jsonOperation = jsonOperation;

}

@Override

public Document toDocument(AggregationOperationContext aggregationOperationContext) {

return aggregationOperationContext.getMappedObject(Document.parse(jsonOperation));

}

}

Now that we have the ability to pass a custom JSON query into our mongodb spring implementation, all that is left is to plug those values into a [TypedAggregation](https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/core/aggregation/TypedAggregation.html) query.

public List<FulfillmentChannel> getFulfillmentChannels(

String SOME\_VARIABLE\_STRING\_1,

String SOME\_VARIABLE\_STRING\_2) {

AggregationOperation match = Aggregation.match(

Criteria.where("dayOfWeek").is(SOME\_VARIABLE\_STRING\_1));

AggregationOperation match2 = Aggregation.match(

Criteria.where("deliveryZipCodeTimings").ne(Collections.EMPTY\_LIST));

String query =

"{ $lookup: { " +

"from: 'deliveryZipCodeTiming'," +

"let: { location\_id: '$fulfillmentLocationId' }," +

"pipeline: [{" +

"$match: {$expr: {$and: [" +

"{ $eq: ['$fulfillmentLocationId', '$$location\_id']}," +

"{ $eq: ['$zipCode', '" + SOME\_VARIABLE\_STRING\_2 + "']}]}}}," +

"{ $project: { \_id: 0, zipCode: 1, cutoffTime: 1 } }]," +

"as: 'deliveryZipCodeTimings'}}";

TypedAggregation<FulfillmentChannel> aggregation = Aggregation.newAggregation(

FulfillmentChannel.class,

match,

new CustomProjectAggregationOperation(query),

match2

);

AggregationResults<FulfillmentChannel> results =

mongoTemplate.aggregate(aggregation, FulfillmentChannel.class);

return results.getMappedResults();

}

At Universe, we use Mongo as a primary data store for our events, users, sessions, and more. The documents we store in our collections tend to be mostly uniform attribute-wise. We model relationships using the standard relationship\_id approach implemented by most ORM/ODMs like Mongoid (a Ruby library).

To date, we’ve relied on the domain modelled in our application — a consumer-facing Rails app — to implement traversals of relationships between documents. Mongoid provides us with easy-to-use relationship helpers that makes our best code concise and expressive.

Alas, the tale often told about ORM/ODMs is true: sometimes you just need to write a query. But wait, isn’t querying for relationships going to be a pain? Aha! It doesn’t have to be if you use the Mongo Aggregation Pipeline.

**What are aggregations?**

*Aggregation process documents and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.*

Aggregations can be used to apply a sequence of query-operations to the documents in a collection, reducing and transforming them. With aggregations, we can perform queries that offer similar functionality to the behaviors we might expect to see in a relational database query.

In Mongo, aggregations work as a [**pipeline**](https://docs.mongodb.com/manual/core/aggregation-pipeline/), or a list of operators/filters applied to the data. Operators come in three varieties: [stages](https://docs.mongodb.com/manual/reference/operator/aggregation/match/#stage-operators), [expressions](https://docs.mongodb.com/manual/reference/operator/aggregation/match/#expression-operators), and [accumulators](https://docs.mongodb.com/manual/reference/operator/aggregation/match/#accumulators). When calling aggregate on a collection, we pass a list of stage operators. Documents are processed through the stages in sequence, with each stage applying to each document individually.

**$match (where)**

$match is a stage operator with this definition: { $match: { <query> } }

The syntax for query is identical [read operation query](https://docs.mongodb.com/manual/tutorial/query-documents/#read-operations-query-argument) syntax. Ideally, you will use $match as early in the pipeline as possible:

*Because*$match*limits the total number of documents in the aggregation pipeline, earlier*$match*operations minimize the amount of processing down the pipe.*

In effect, with $match Mongo will filter the collection accoring to the query parameters, and only pass through the *documents matching the query*, to the next stage of the pipeline. Take this example of “article” documents in a collection, with an author — and then filtering by the name of the author:

We can apply other conditions and constraints by using an expression operator (one of the two other operators in addition to stage, mentioned above). For example, we can pass $or to $match and then provide a list of matchable queries, one of which must be true in order for the document to be included by the filter. This is an example of a [boolean aggregation](https://docs.mongodb.com/manual/reference/operator/aggregation-boolean/" \t "_blank).

We can also apply additional constraints such as [comparisons](https://docs.mongodb.com/manual/reference/operator/aggregation-comparison/), where document values must be greater than, less than, or equal to some other value. Here we use both $or and $gte in a $match stage:

**$skip + $limit**

$skip and $limit both accept a positive integer — and do what you expect them to do: skip documents, and limit the number returned. For the purposes of developing rich APIs on top of Mongo, this is obviously quite useful. A simple example building on the above example of an articles collection might look like:

db.articles.aggregate([  
 { $match: { score: { $gt: 60 }}},  
 { $limit: 2},  
 { $skip: 1 }  
])

First we’re filtering the collection for documents with a field called score with a value greater than 60. Next, we limit the number of documents to two, and then return the second one by skipping the first.

$skip and $limit are also subject to [pipeline optimizations](https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/).

**$lookup (join)**

Ok, so far — so good. But what about modelling relationships?

$lookup kind of functions the way a join does — it matches each document in the pipeline stage to a set of documents from another collection, and then returns those documents *as an attribute on the current one* to the next stage of the pipeline. [You can even match arrays](https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/#use-lookup-with-an-array).

The syntax of this operation looks like:

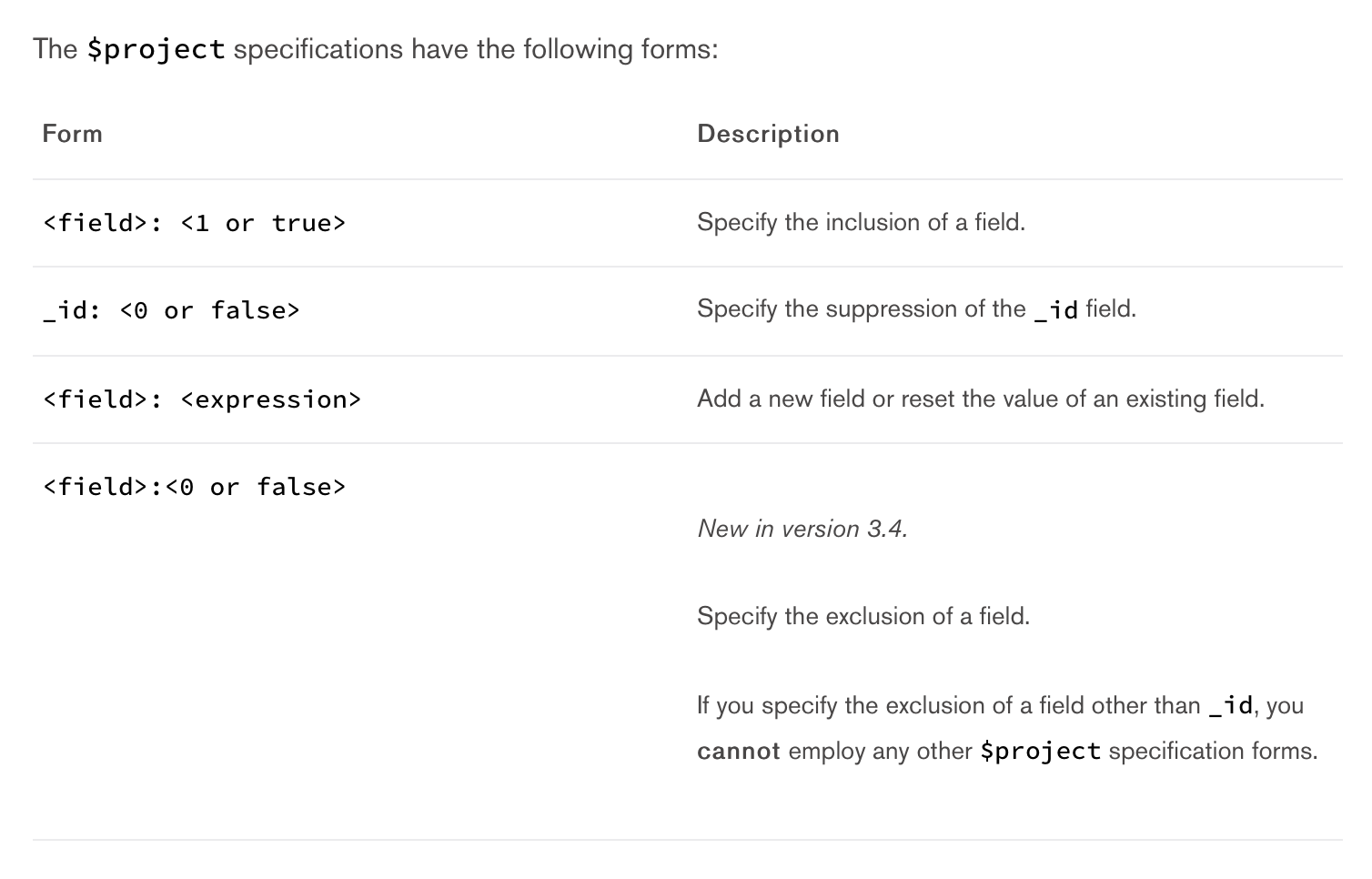
{ $lookup: {  
 from: <collection to join>,  
 localField: <field from the input documents>,  
 foreignField: <field from the documents of the "from" collection>,  
 as: <output array field>  
} }

As a full fledged example inside of database modelling an inventory system:

**$project (select)**

Naturally, we’re going to want to reduce the documents into smaller objects — returning just the fields we want, or aliasing their names. In the SQL paradigm, this sounds like a SELECT , for Mongo it’s $project .

The structure you pass to $project is a field mapping:



We can actually compose very complex serialization routines using $projectwith <expression> . Otherwise, including a field is as easy as passing <field>: <1 or true> . The documents returned to the next stage of the pipeline *will only contain* the values specified by $project.

Hopefully you’re excited to try out aggregations yourself! The current version of the Mongo manual (3.4) provides some excellent example datasets to experiment with: