# Implementation decisions

I used the Rest template to access the underlying apis because there is no internal business logic with regard to orders, shipments and the like and because manipulating the api results to create an aggregated result in the required format proved too complex when using Spring Web Flux.

The Aggregation requests are handled by the Aggregation Controller. I always try to delegate as much as possible to other classes in order to keep controllers small.

I created Service objects that use the Rest template to query the Shipment, Pricing and Track API’s. This is to keep the design clean and the code in the AggregationController small. Using these it was straightforward to achieve task 1.

For tasks 2 and 3, I created an abstract service adapter class that is accessed by the AggregationController instead of the api Services. The adapter takes care of the bulk requests and the cache and queue. Each API service has its own adapter. The logic for the adapters is the same, the only differences are the queue and cache they use and the service they call.

The adapter accesses a ServiceCache which actually contains both the queue for request ids and a Map containing results already retrieved, per id. The queue is implemented as a LinkedBlockingQueue. Each of the three adapters has its own ServiceCache. The ServiceCaches are configured in the bean configuration as singletons, so that multiple instances of adapter access the same cache and queue. Every controller instance has its own set of three adapters, but all adapters for for instance the ShipmentService need to access the same queue and cache.

Note that when an adapter cannot immediately return all the needed results for the incoming request, it blocks until the cache contains all the needed results (after other bulk requests have added them), or until 5 seconds have passed (task 3).

The adapter is ‘helpful’, in the sense that upon an incoming request, it will first check if the queue is full and eventually perform the bulk request, and then check for its own results in the cache. That last bulk request may just deliver the last needed results for this request.

When emptying the queue to perform the bulk request, the thread locks the queue as the drain operation is not thread-safe.

After performing the bulk request, it adds its own ids to the queue per id, if there is room and there is not already a result in for that id. If no room, it tries again on the next iteration of the loop.

The controller uses an Executor to call the three adapters simultaneously and then gathers the results and returns them.

The ServiceCache has a method to insert ids into the queue that checks if an id is not already on the queue.

In hindsight: the ServiceCache breaks encapsulation because we often access the queue and result map directly, but it is a container object that makes sure the cache and queue are unique per adapter while not keeping a lot of code in a singleton.

Also, Ido not seem to need the blocking aspect of the put() function of the BlockingLinkedQueue in my code as I check the size of the queue beforehand.