Marketplace project description

Nota Bene

Get list of all user's orders

We need to show all user's orders with bought products and download links for each product. I found several solutions:

- Get all user's orders, then for each order get order item. Obvious that it is not good solution because we have unpredictable number of database queries.
- Use OneToMany relationship. It looks better, but we can suggest that we don't know how many data will be pulled cascaded from DB (other tables depended on `order` may have OneToMany relationship).
- Get all order items with one query and work with them. But we will need to groupe them in some way.

The last solution was chosen. We get all order items then map them to orders. You can found that `OrderItem` have `@OneToMany` relationship to `download`, we admit that because `download` table have only one relationship (with `order_item`).

Final sql generated by `OrderItemDaoImp.find()` looks like this:

```sql
SELECT
  order_item.id,
  product.id,
  book.id,
  "order".id,
  download.id
  -- Other data from all tables
FROM
  order_item
  LEFT OUTER JOIN product ON order_item.product_id = product.id
  LEFT OUTER JOIN book ON product.id = book.id
  LEFT OUTER JOIN "order" ON order_item. "order_id" = "order".id
  LEFT OUTER JOIN download ON order_item.id = download.order_item_id
WHERE
  "order"."user_id" = 1;

```

Convert from `OrderItem` to `List<Order>`:

```java
@Override public List<IOrder> getOrdersByUserId(final Integer userId) {
  final List<IOrderItem> orderItemList =
orderItemService.getOderItemsByUserId(userId);

  final Set<IOrder> orderSet = new HashSet<>();
  orderItemList.forEach(orderItem ->
    orderSet.add(orderItem.getOrder().addOrderItem(orderItem))
  );

  final List<IOrder> orderList = new ArrayList<>(orderSet);
  orderList.sort((o2, o1) -> o1.getCreated().compareTo(o2.getCreated()));
  return orderList;
}
```

We use `Set` to avoid to search proper **order** in list, on the other hand - we need to convert them to sorted list.

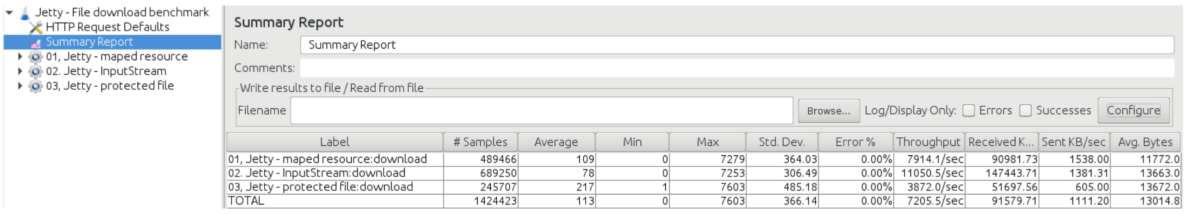
Download benchmark (public vs static files)

Test on Jetty

We need to chouse how to server static files. Let's consider two solution: serving by servlet and serving by our app as InputStream.

We perform three test:

1. access file from servlet (we will use Jetty)
2. access file from our app
3. access protected file from app (we will need to make database query to get the file)



Jetty - File download benchmark
HTTP Request Defaults
Summary Report
01, Jetty - maped resource
02. Jetty - InputStream
03, Jetty - protected file
Browse...
Configure

Summary Report

Name: Summary Report

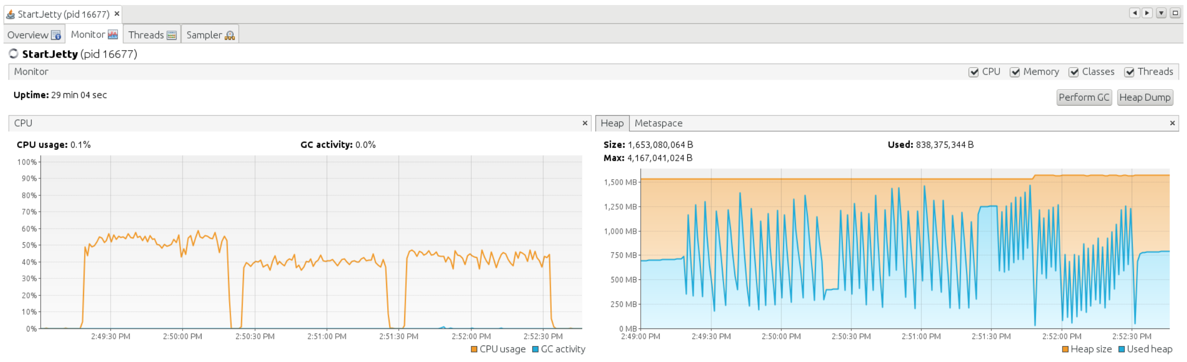
Comments:

Write results to file / Read from file

Filename

Log/Display Only: ☐ Errors ☐ Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
01, Jetty - maped resource:download	489466	109	0	7279	364.03	0.00%	7914.1/sec	90981.73	1538.00	11772.0
02. Jetty - InputStream:download	689250	78	0	7253	306.49	0.00%	11050.5/sec	147443.71	1381.31	13663.0
03, Jetty - protected file:download	245707	217	1	7603	485.18	0.00%	3872.0/sec	51697.56	605.00	13672.0
TOTAL	1424423	113	0	7603	366.14	0.00%	7205.5/sec	91579.71	1111.20	13014.8



StartJetty (pid 16677)

Uptime: 29 min 04 sec

CPU usage: 0.1%

GC activity: 0.0%

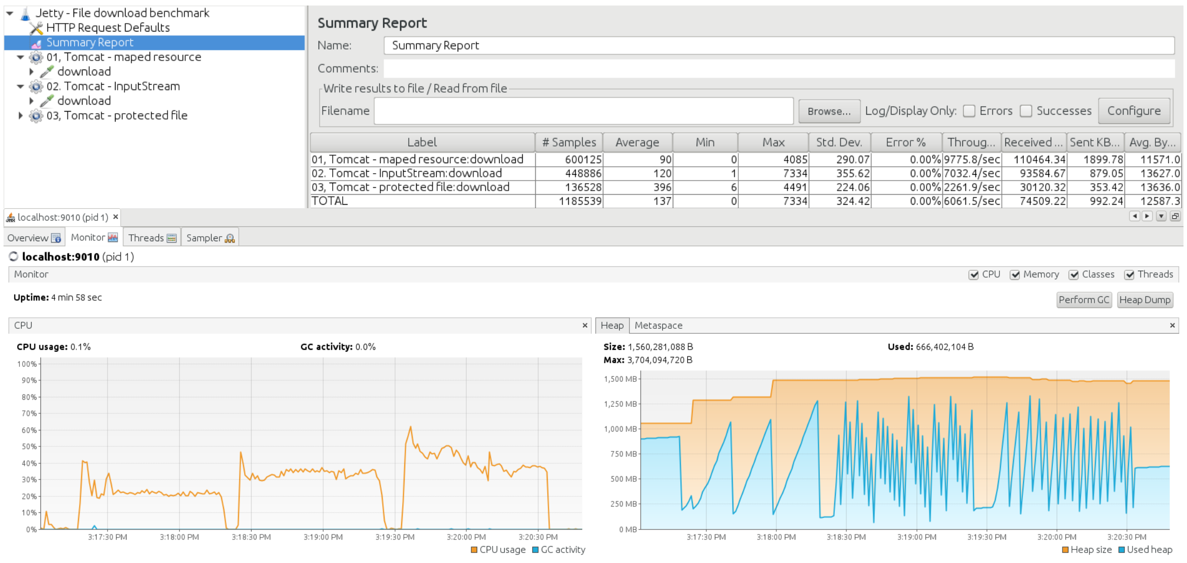
Size: 1,653,080,064 B

Max: 4,167,041,024 B

Used: 838,375,344 B

As we can see we have similar resource usage, and response time for both solutions. Even Jetty show little worse performance then our app. So let's test it on Tomcat.

Test on Tomcat


Jetty - File download benchmark
HTTP Request Defaults
Summary Report
01, Tomcat - maped resource
download
02. Tomcat - InputStream
download
03, Tomcat - protected file

Summary Report
Name: Summary Report
Comments:
Write results to file / Read from file
Filename
Log/Display Only: ☐ Errors ☐ Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throug...	Received ...	Sent KB...	Avg. By...
01, Tomcat - maped resource:download	600125	90	0	4085	290.07	0.00%	9775.8/sec	110464.34	1899.78	11571.0
02. Tomcat - InputStream:download	448886	120	1	7334	355.62	0.00%	7032.4/sec	93584.67	879.05	13627.0
03, Tomcat - protected file:download	136528	396	6	4491	224.06	0.00%	2261.9/sec	30120.32	353.42	13636.0
TOTAL	1185539	137	0	7334	324.42	0.00%	6061.5/sec	74509.22	992.24	12587.3

localhost:9010 (pid 1)
localhost:9010 (pid 1)
Monitor ✓ CPU ✓ Memory ✓ Classes ✓ Threads
Uptime: 4 min 58 sec
CPU usage: 0.1% **GC activity:** 0.0%
Size: 1,560,281,088 B **Used:** 666,402,104 B
Max: 3,704,094,720 B

OK. On Tomcat we got expected results. But why access to protected file took two times as long as access on Jetty? I don't know.

As conclusion we can say that both solutions are very slow and resource hungry. Java servlets are not suitable for file serving, we should use proper tools like Nginx for that.