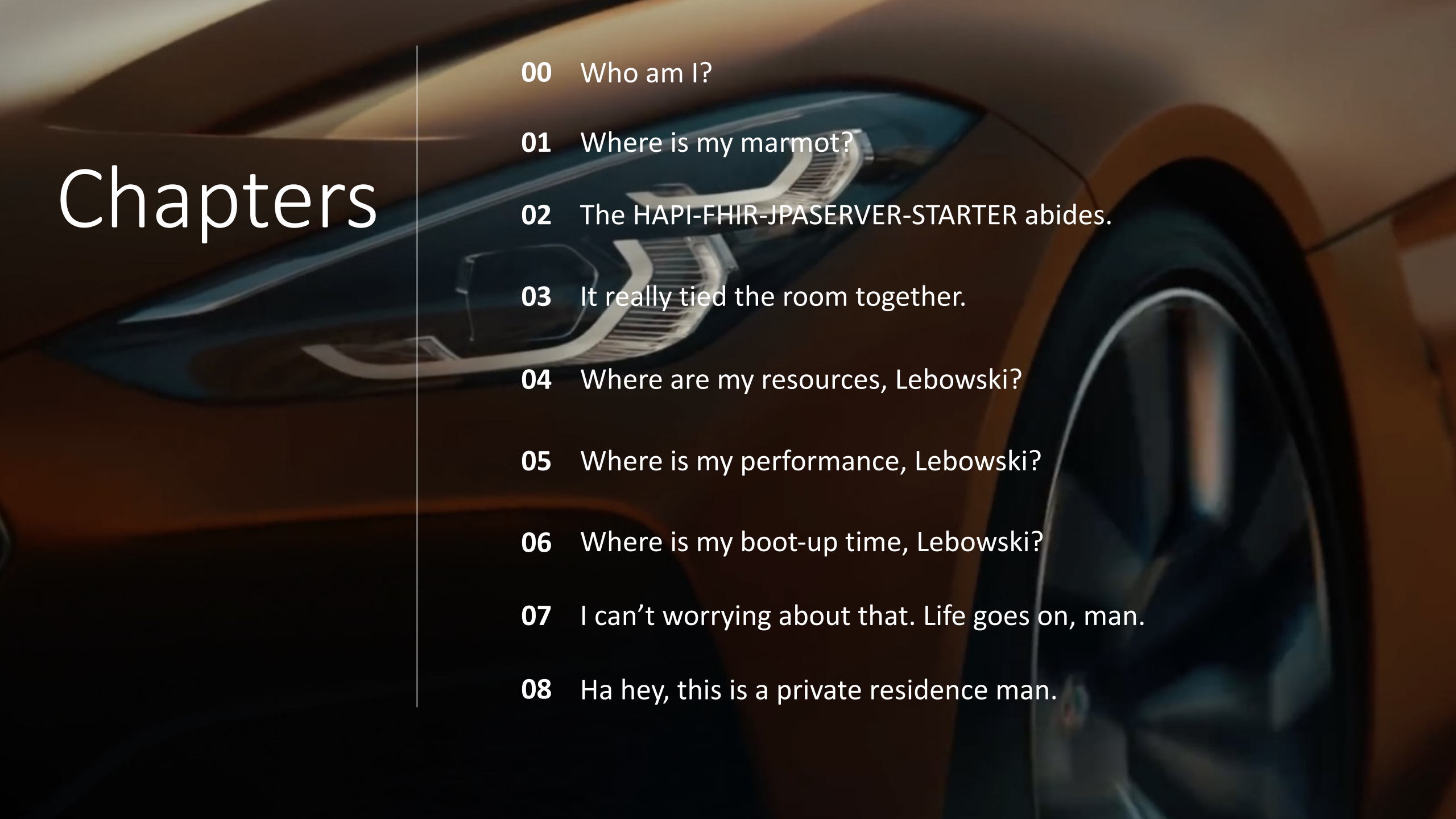
HAPI FHIR JPA Server Starter Performance Investigations

HAPI FHIR JPA Server Tuning

December 2021



Chapters

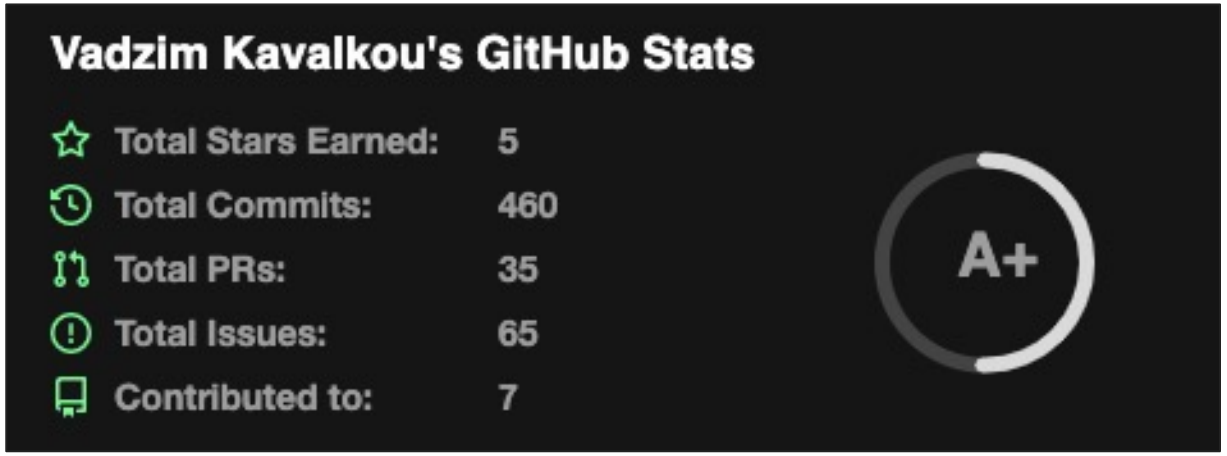
Who am I?

VADZIM KAVALKOU

Lead Software Engineer at EPAM Systems with more than 5 years of IT experience.

Key business domains: Finances; Human Resources, HealthCare

A T-shaped person with core knowledge in the JVM stack area. Proven practical experience with development with Java (up to 17), Kotlin, Spring Boot (up to 2.6.1), JIB, Docker, Open API, Gradle, Postgres (up to 14), Flyway/Liquibase. Familiar with ReactJS, Kubernetes, GitHub/Gitlab CI, JMeter performance testing, Google Cloud Development, and Architecting.



Where is my marmot?

Topics:

- HL7
- HAPI FHIR

HL7

What is HL7 FHIR?

FHIR – Fast Healthcare Interoperability Resources – is a next generation standards framework created by HL7. FHIR combines the best features of HL7's v2 , HL7 v3 and CDA product lines while leveraging the latest web standards and applying a tight focus on implementability.

FHIR solutions are built from a set of modular components called "Resources". These resources can easily be assembled into working systems that solve real world clinical and administrative problems at a fraction of the price of existing alternatives. FHIR is suitable for use in a wide variety of contexts – mobile phone apps, cloud communications, EHR-based data sharing, server communication in large institutional healthcare providers, and much more.

```
<Patient xmlns="http://hl7.org/fhir">
  <id value="glossy"/>
  <meta>
    <lastUpdated value="2014-11-13T11:41:00+11:00"/>
  </meta>
  <text>
    <status value="generated"/>
    <div xmlns="http://www.w3.org/1999/xhtml">
      <p>Henry Levin the 7th</p>
      <p>MRN: 123456. Male, 24-Sept 1932</p>
    </div>
  </text>
  <extension url="http://example.org/StructureDefinition/trials">
    <valueCode value="renal"/>
  </extension>
  <identifier>
    <use value="usual"/>
    <type>
      <coding>
        <system value="http://hl7.org/fhir/v2/0203"/>
        <code value="MR"/>
      </coding>
    </type>
    <system value="http://www.goodhealth.org/identifiers/mrn"/>
    <value value="123456"/>
  </identifier>
  <active value="true"/>
  <name>
    <family value="Levin"/>
    <given value="Henry"/>
    <suffix value="The 7th"/>
  </name>
  <gender value="male"/>
  <birthDate value="1932-09-24"/>
  <careProvider>
    <reference value="Organization/2"/>
    <display value="Good Health Clinic"/>
  </careProvider>
</Patient>
```

Resource Identity & Metadata

Human Readable Summary

Extension with URL to definition

Standard Data:
- MRN
- Name
- Gender
- Birth Date
- Provider

HAPI FHIR

HAPI FHIR is a complete implementation of the HL7 FHIR standard for healthcare interoperability in Java. They are an open community developing software licensed under the business-friendly Apache Software License 2.0. HAPI FHIR is a product of Smile CDR.

The HAPI-FHIR-JPASERVER-STARTER abides.

Topics:

- HAPI-FHIR-JPA-SERVER-STARTER
- DATABASE SCHEMA
- PROS AND CONS

HAPI-FHIR-JPASERVER-STARTER

The HAPI FHIR team provides a set of open source solutions for HL7 FHIR. One of them is hapi-fhir-jpaserver-starter.

So, what is the *hapi-fhir-jpaserver-starter* (HFJS)?

This project is a complete starter project you can use to deploy a FHIR server using HAPI FHIR JPA. This is a typical solution based on Spring Boot.

To check up on what you can bring useful in your project, please visit the demo server. It's absolutely not necessary to use all the resources, you can move to microservice architecture and split logic for different domain-based areas.

This project is a fully contained FHIR server, supporting all standard operations (read/create/delete/etc). It bundles an embedded instance of the H2 Java Database so that the server can run without depending on any external database, but it can also be configured to use an installation of Oracle, Postgres, etc.

6.2 HAPI FHIR JPA Architecture

The HAPI JPA Server has the following components:

- **Resource Providers:** A RESTful server Resource Provider is provided for each resource type in a given release of FHIR. Each resource provider implements a @Search method implementing the complete set of search parameters defined in the FHIR specification for the given resource type.

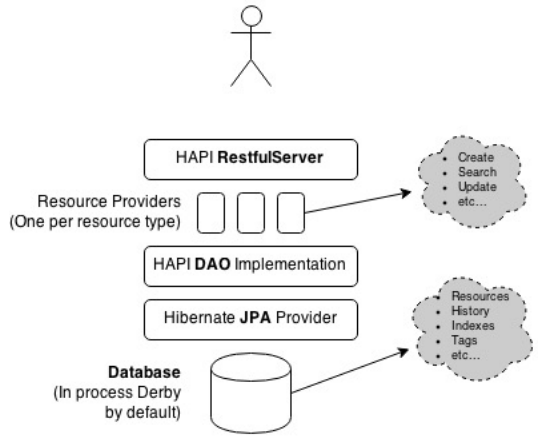
  The resource providers also extend a superclass which implements all of the other FHIR methods, such as Read, Create, Delete, etc.

  Note that these resource providers are generated as a part of the HAPI build process, so they are not checked into Git. The resource providers do not actually implement any of the logic in searching, updating, etc. They simply receive the incoming HTTP calls (via the RestfulServer) and pass along the incoming requests to the DAOs.

- **HAPI DAOs:** The DAOs actually implement all of the database business logic relating to the storage, indexing, and retrieval of FHIR resources, using the underlying JPA API.

- **Hibernate:** The HAPI JPA Server uses the JPA library, implemented by Hibernate. No Hibernate specific features are used, so the library should also work with other providers (e.g. Eclipselink) but it is not tested regularly with them.

- **Database:** The RESTful server uses an embedded Derby database, but can be configured to talk to any database supported by Hibernate.

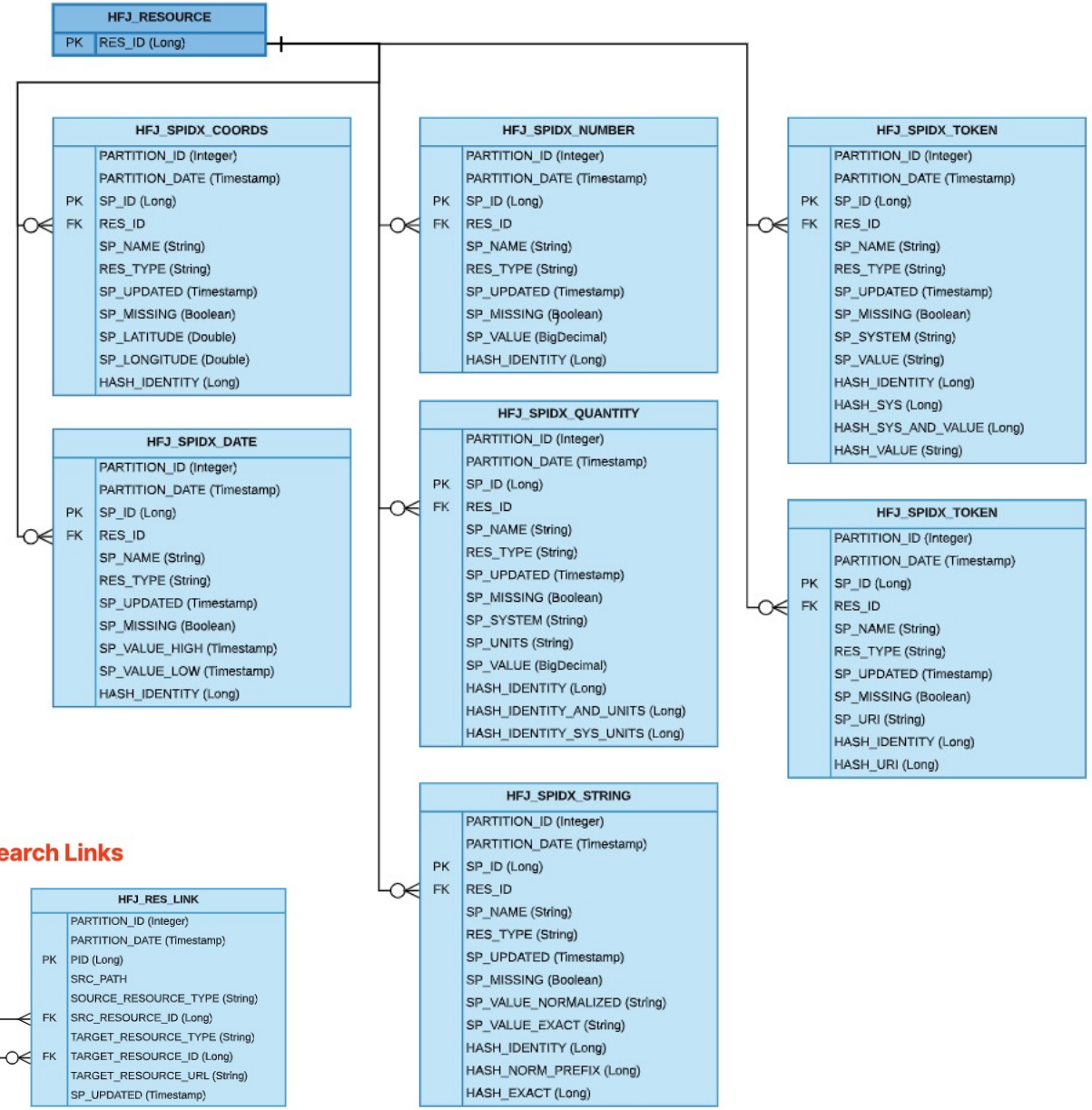


DATABASE SCHEMA

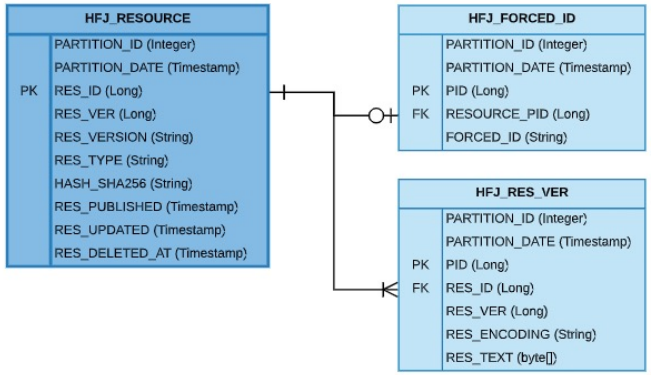
The HAPI FHIR JPA schema relies heavily on the concept of internal persistent IDs on tables, using a Java type of Long (8-byte integer, which translates to an *int8* or *number(19)* on various database platforms).

Many tables use an internal persistent ID as their primary key, allowing the flexibility for other more complex business identifiers to be changed and minimizing the amount of data consumed by foreign key relationships. These persistent ID columns are generally assigned using a dedicated database sequence on platforms which support sequences.

The persistent ID column is generally called PID in the database schema, although there are exceptions.



6.3.2 HFJ_RESOURCE: Resource Master Table



6.3.5 HFJ_RES_LINK: Search Links

The HFJ_RESOURCE table indicates a single resource of any type in the database. For example, the resource `Patient/1` will have exactly one row in this table, representing all versions of the resource.

PROS AND CONS

To be honest, the HL7 and HAPI FHIR is not the easiest architecture and solution to understand. You will need some patience, concentration, and time to feel strong in the healthcare domain. It has one great killer feature: during the start of the application all the schemas and database structures are created as it should be done for HL7. You can bring your custom changes easily using autoconfiguration, tune your database settings, such as indexes, resource validations both for requests and responses, types of supported resources. In spite of all the benefits that HFJS brings to us, there are a lot of trade-offs. And in a world where every millisecond may cost you a huge profit, the performance, boot-up time, and resources consumption have a great impact.

Taking all this into consideration, I would like to define pros and cons of HAPI-FHIR-JPA-SERVER:

PROS:

1. Out of the box solution;
2. Open-source and supported by community;
3. Automated schema creation.

CONS:

1. Out of the box solution:
   1.1. Hard to understand the sources and database schema;
2. Low performance;
3. Huge boot-up time;
4. Vast resources consumption;
5. Is not flexible:
   2.1. Not able to be tuned in different application layers;
   2.2. Not able to support spring-indexer and resilience4j;

IT REALLY TIED THE ROOM TOGETHER.

Topics:

- DEFAULT STABLE SETUP
- SPRING BOOT CONFIGURATION
- PERFORMANCE TESTS DESCRIPTION

DEFAULT STABLE SETUP

After a few months of investigations and data preparations for HFJS, we get the vision of possible resources consumption. Before tuning our configuration that could be stable without facing OOM both on JVM and PostgreSQL is the next:

JDK	GC	SERVER	GRADLE	POSTGRES	POSRGRES VM RAM	POSRGRES VM CORE	POSTGRES VM SSD	JVM RAM	JVM CORE
11	G1	Tomcat	7.1	13.4	32 GiB	8	2 TB	16 GiB	4

We've took a useful for us parts of HFJSS, such as hapi-fhir-base, hapi-fhir-jpaserver-base, hapi-fhir-validation, hapi-fhir-structures-r4, hapi-fhir-validation-resources-r4. And we build our own theme park, with blackjack. The versions of used libs:

HAPI FHIR VERSION	HIKARI VERSION	SPRING VERSION	SPRING CLOUD VERSION
5.5.2	5.0.0	2.5.6	2020.0.4

SPRING BOOT CONFIGURATION

The basic configuration for HFJSS had been tuned a bit, mostly tuning affected pools and workers, some searching, validation, and reindexing settings.

If you are not a lot familiar with hikari pool tuning, there is a great article about pool sizing.

One important thing is that we disabled hibernate caching to get clear performance metrics of the server as it is.

The average time to boot up the service depends on if it's the first start because the database structure could be created or not. For sure, on production, we will use warm-up, and mostly we will have the situation without creating a database structure. We will cover this and other topics in next chapters.

To my mind, I ought to share with you some use cases that we will cover with performance tests. You can find that most of all there are READ operations, and that is correct. The user will do these operations most of all, and we should follow his/her behavior.

PERFORMANCE TESTS DESCRIPTION

Outpatient practice (ambulatory):

#	Scenario step	Assumed FHIR data operation
1	Login	
2	As a doctor, I want to open my patient's medical card with general information page with diagnoses	Get Patient resource, get several (10) Condition resources for this patient
3	As a doctor, I want to look through titles and dates of all available to me medical records (sections) for my patient for all the time	
4	As a doctor, I want to open some of them and read (e.g. General blood analysis 12.09.2003, Renal ultrasound 18.02.2014, General blood analysis 22.06.2020). Alternative flow: As a doctor, I want to scroll and look through all sections in my patient's card	Simultaneously get 10 DiagnosticReports by IDs, including all references recursively (_include:recurse=*)
5	As a doctor, I want to create a new medical record (section) of today's visit and save it	Create DiagnosticReport with subject = this patient and 10 Observations in a result array
6	As a doctor, I want to close current patient and see a list of patients again as described in (2)	See (2)
7	Repeat (2) - (7) every 15-30 minutes for each patient	
8	Logout	

PERFORMANCE TESTS DESCRIPTION

Inpatient practice (hospital):

#	Scenario step	Assumed FHIR data operation
1	Login	
2	As a doctor, I want to see my list of patients, that are in my responsibility today: patients that are now in my hospital in my department and assigned to me	Find all Patients, that have EpisodeOfCare with period including [some date] AND [some] managingOrganization
3	As a doctor, I want to open my patient's medical card with general information page	Get Patient resource, get several (5-20) resources for this patient, like AllergyIntolerance, Condition, RelatedPerson, Immunization
4	As a doctor, I want to look through titles and dates of all available to me medical records (sections) for my patient for all the time	Select all Compositions with subject = this patient (in average, 50, can be 5-200)
5	As a doctor, I want to open some of them and read (e.g. General blood analysis 12.09.2003, Renal ultrasound 18.02.2014, General blood analysis 22.12.2021)	For each of 3 opened medical records: Get Composition, in it's section for all entryes select DiagnosticReport, in it's result for all references select all (1-25) Observations
6	As a doctor, I want to look through dates and diagnoses of all hospitalizations of my patient	Find all EpisodeOfCare that was for this Patient
7	As a doctor, I want to open some hospitalizations (medical forms) and read all sections in them (e.g. Acute appendicitis 21.04.2017-25.04.2017, Pneumonia 13.09.2019-28.09.2019)	For each hospitalization (EpisodeOfCare) – find Composition that in event in detail contains a reference to current EpisodeOfCare. For that Composition for all sections (5-20) in it's entryes select all (1-25) Observations
8	As a doctor, I want to open a medical form for current hospitalization and read all sections	The same as (7)
9	As a doctor, I want to create a new section in this medical form with a daily medical record of my examination	Create Composition with subject = this patient and 10-30 Observations as entryes in it's section
10	As a doctor, I want to close current patient and see a list of patients again as described in (2)	See (2)
11	Repeat (2) - (7) every 15-30 minutes for each patient	
12	Logout	

WHERE ARE MY RESOURCES, LEBOWSKI?

RESOURCES PREDICTION

Moreover, it would be nice to provide you with some information regarding the datasets we've used.

Due to the fact, that we are not going to disclose our ambitions and share some business ideas I will bring you the high-level information about the resources amount.

We will take in use 4 steps with almost x2 scaling of resources amount in our database. These steps will be reminded on the next slides.

WHERE IS MY PERFORMANCE, LEBOWSKI?

Topics:

- INTRODUCTION
- TOMCAT RESULTS
- JETTY RESULTS
- UNDERTOW RESULTS
- EMBEDDED SERVERS' COMPARISION

INTRODUCTION

At the very beginning of this chapter, I would like to share with you some special secrets. The hapi fhir server is a healthcare one and the business is not going to deal with a huge RPS. But the metrics that I will share with you soon could help us to understand the possible scaling strategy better, follow fault-tolerance best practices, and provide the vision of resources consumption. The last but not least, all these metrics can help us to predict some cost management.

Let's assume that the average institution has the biggest load from 8:00 till 17:00. As a result, we will have ~40 business hours per week. The next metrics will be presented for a typical business day (transactions per working day, TPWD in future) and RPS as well. And now, show me the numbers, dude!

TOMCAT RESULTS

STEP	TRANSACTIONS PER WORKING DAY	TRANSACTIONS PER SECOND	RESPONSE TIMES OVER TIME	HITS PER SECOND	NODE CPU	NODE RAM (GiB)	JVM RATE (MAX ops/s)	JVM HEAP (GiB)	JVM NON-HEAP (Mib)	JVM CPU (%)	THREADS (MAX)	HIKARI POOL (MAX)	HIKARI CONNECTIONS TIME (MAX ms)	POSTGRES STATEMENTS CALLS (ops/s)
1	460800	16	~100	~35	~0.400	~5	~31	~3,20	~275	~20	~94 (25 runnable, 40 waiting, 45 timed-waiting)	9 active, 15 idle	38 usage, 62 creation	max: 475, avg: 185
2	403200	14	~100	~35	~0.360	~3,36	~31	~2,23	~256	~14	~94 (22 runnable, 40 waiting, 45 timed-waiting)	5 active, 16 idle	35 usage, 45 creation	max: 499, avg: 248
3	604800	21	~200	~54	~0.600	~2,48	~26	~2,14	~257	~14	~104 (22 runnable, 54 waiting, 40 timed-waiting)	7 active, 19 idle	75 usage, 38 creation	max: 499, avg: 248
4	403200	14	~80	~40	~0.360	~6,65	~31	~4,46	~253	~13	~104 (23 runnable, 40 waiting, 41 timed-waiting)	7 active, 16 idle	78 usage, 36 creation	max: 619, avg: 283

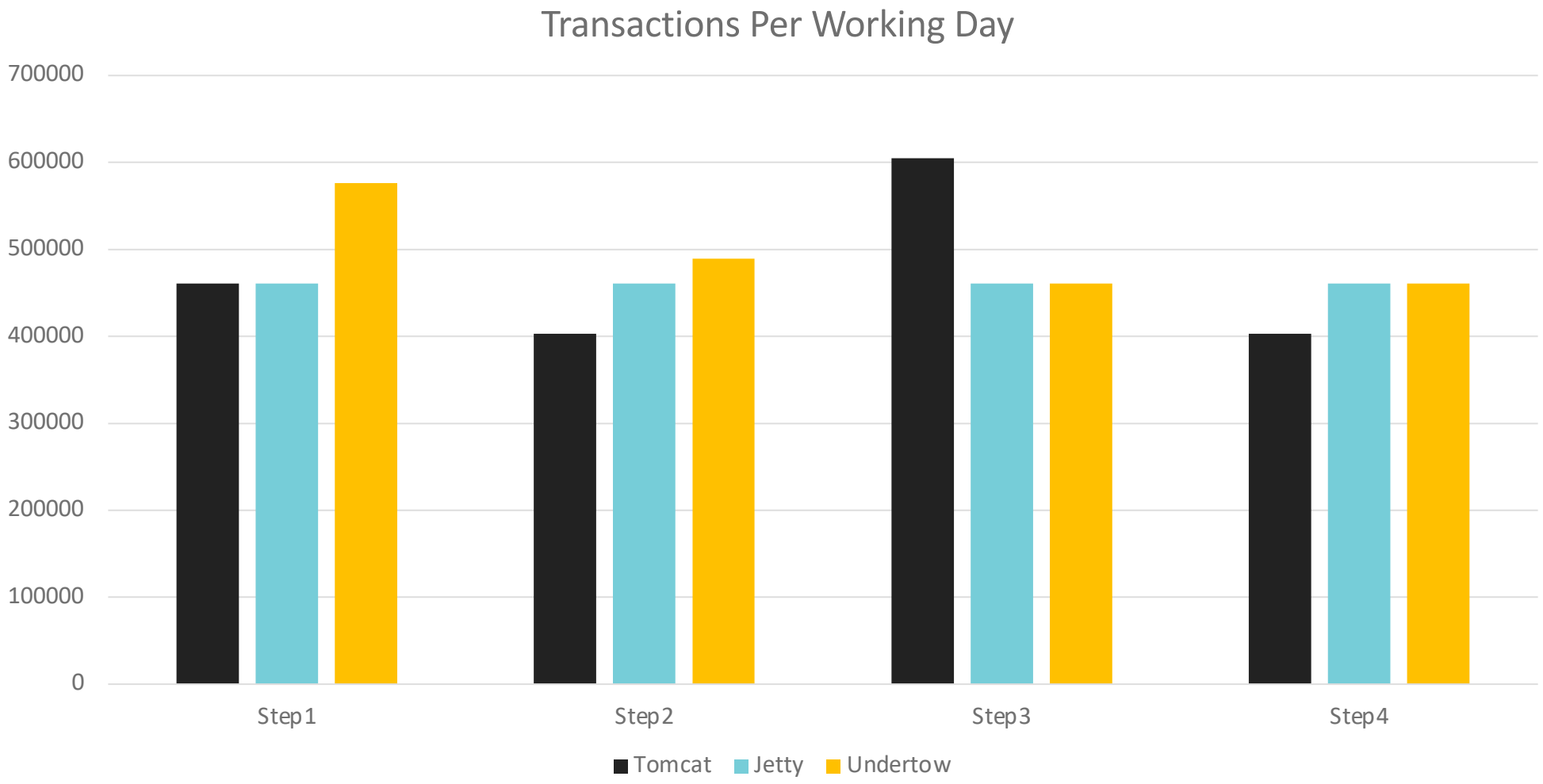
JETTY RESULTS

STEP	TRANSACTIONS PER WORKING DAY	TRANSACTIONS PER SECOND	RESPONSE TIMES OVER TIME	HITS PER SECOND	NODE CPU	NODE RAM (GiB)	JVM RATE (MAX ops/s)	JVM HEAP (GiB)	JVM NON-HEAP (Mib)	JVM CPU (%)	THREADS (MAX)	HIKARI POOL (MAX)	HIKARI CONNECTIONS TIME (MAX ms)	POSTGRES STATEMENTS CALLS (ops/s)
1	460800	16	~100	~40	~0,300	~6	~32	~4,52	~258	~11	~95 (23 runnable, 23 waiting, 54 timed-waiting)	8 active, 16 idle	22 usage, 53 creation	max: 498, avg: 237
2	460800	16	~80	~40	~0,41	~6,01	~31	~4,01	~256	~17	~98 (21 runnable, 23 waiting, 56 timed-waiting)	7 active, 19 idle	18 usage, 125 creation	max: 486, avg: 310
3	460800	16	~80	~40	~0,400	~5,69	~32	~3,34	~261	~19	~88 (24 runnable, 23 waiting, 48 timed-waiting)	6 active, 15 idle	18 usage, 125 creation	max: 515, avg: 262
4	460800	16	~200	~40	~0,350	~3,96	~31	~3,09	~287	~13	~98 (25 runnable, 24 waiting, 58 timed-waiting)	13 active, 16 idle	36 usage, 41 creation	max: 508, avg: 235

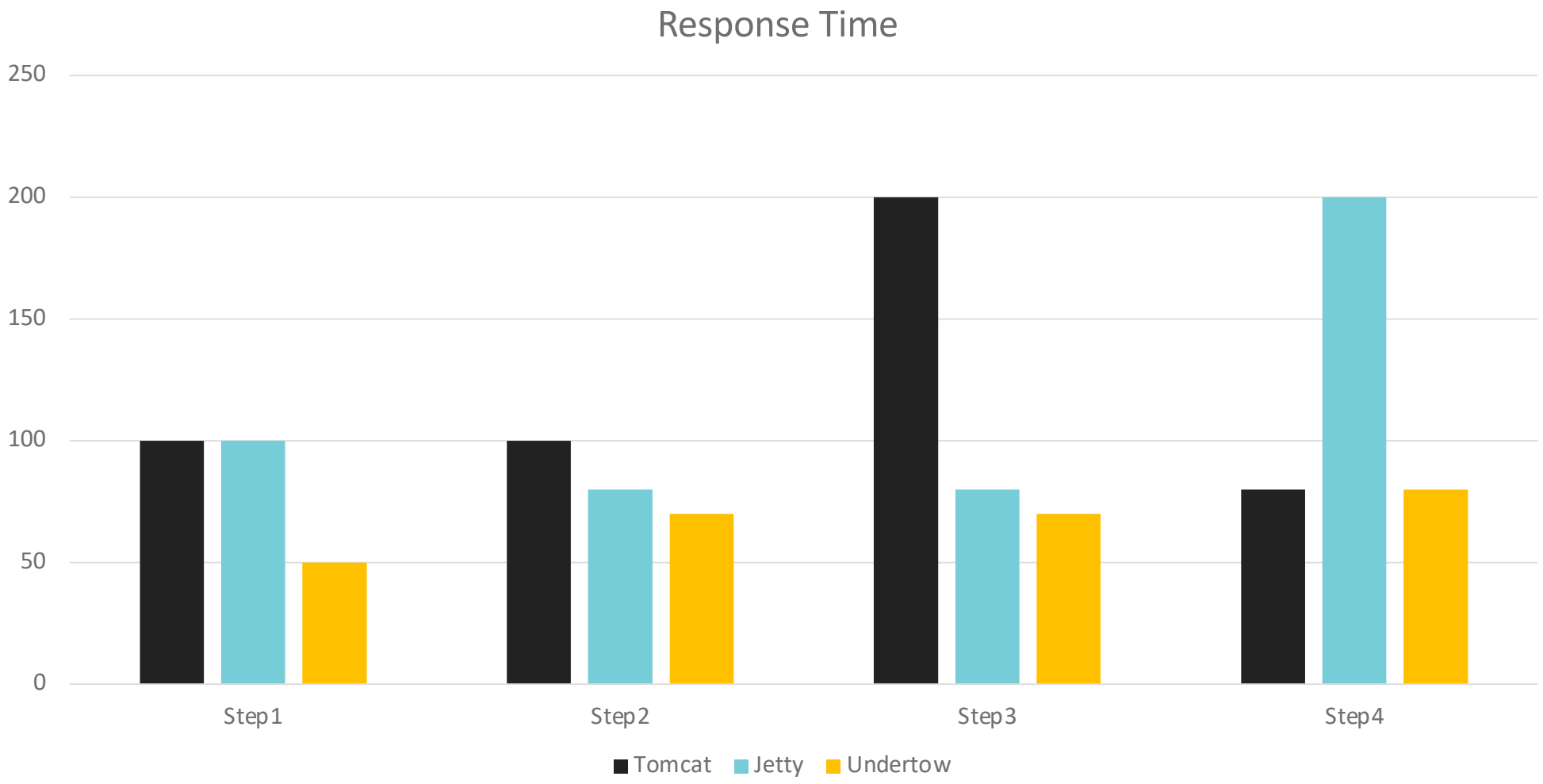
UNDERTOW RESULTS

STEP	TRANSACTIONS PER WORKING DAY	TRANSACTIONS PER SECOND	RESPONSE TIMES OVER TIME	HITS PER SECOND	NODE CPU	NODE RAM (GiB)	JVM RATE (MAX ops/s)	JVM HEAP (GiB)	JVM NON-HEAP (Mib)	JVM CPU (%)	THREADS (MAX)	HIKARI POOL (MAX)	HIKARI CONNECTIONS TIME (MAX ms)	POSTGRES STATEMENTS CALLS (ops/s)
1	576000	18	~50	~43	~0.250	~3,5	~31	~3,83	~256	~10	~89 (27 runnable, 38 waiting, 42 timed-waiting)	7 active, 15 idle	13 usage, 20 creation	max: 684, avg: 350
2	489600	17	~70	~50	~0,360	~3,46	~31	~1,88	~256	~12	~86 (24 runnable, 26 waiting, 39 timed-waiting)	8 active, 19 idle	21 usage, 52 creation	max: 502, avg: 250
3	460800	16	~70	~45	~0,250	~6,51	~31	~4,34	~257	~15	~86 (27 runnable, 10 waiting, 17 timed-waiting)	6 active, 18 idle	103 usage, 55 creation	max: 515, avg: 192
4	460800	16	~80	~40	~0,380	~4,65	~33	~2,57	~251	~28	~87 (27 runnable, 26 waiting, 40 timed-waiting)	7 active, 16 idle	69 usage, 51 creation	max: 523, avg: 307

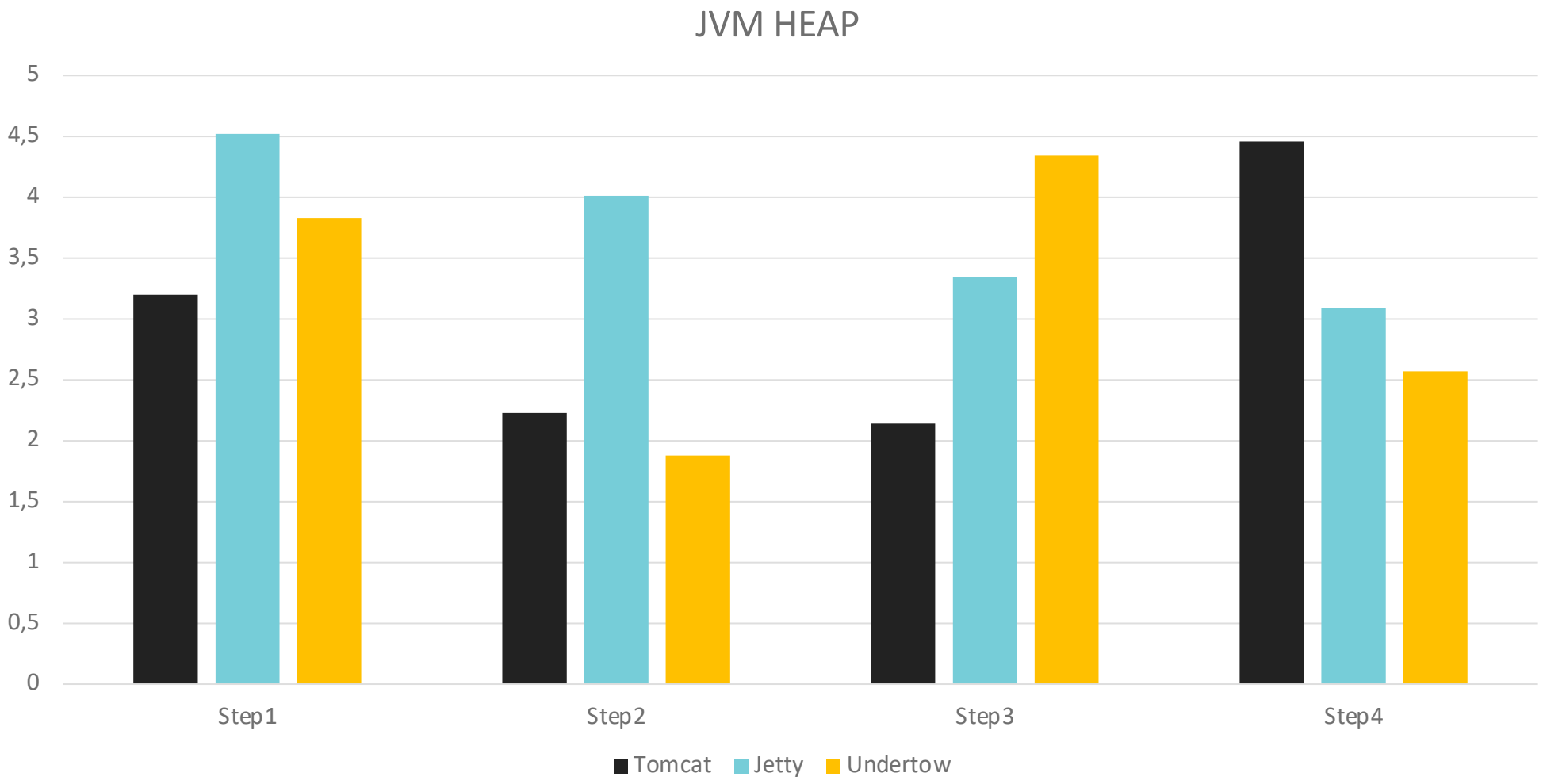
EMBEDDED SERVERS' COMPARISON



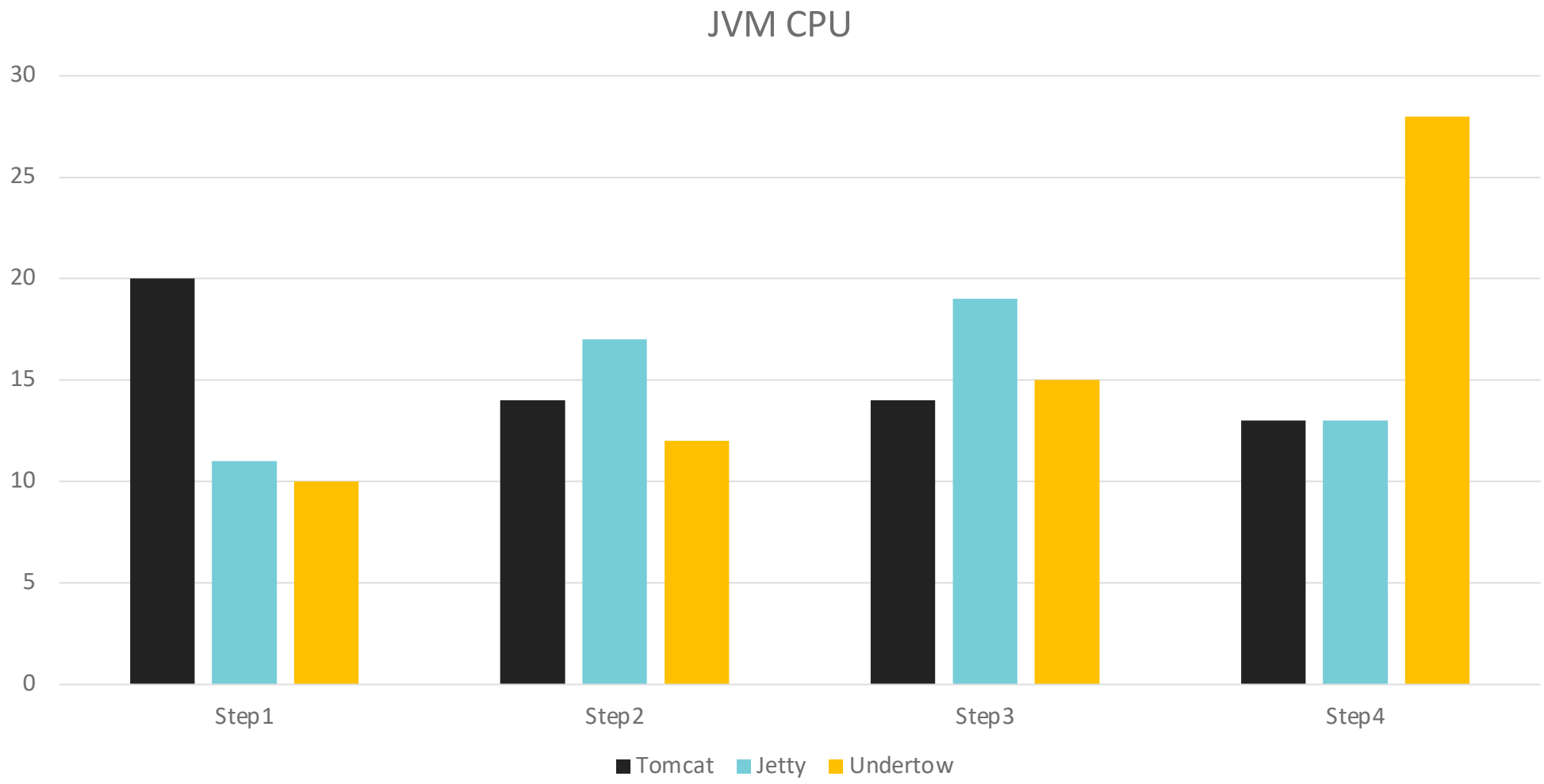
EMBEDDED SERVERS' COMPARISON



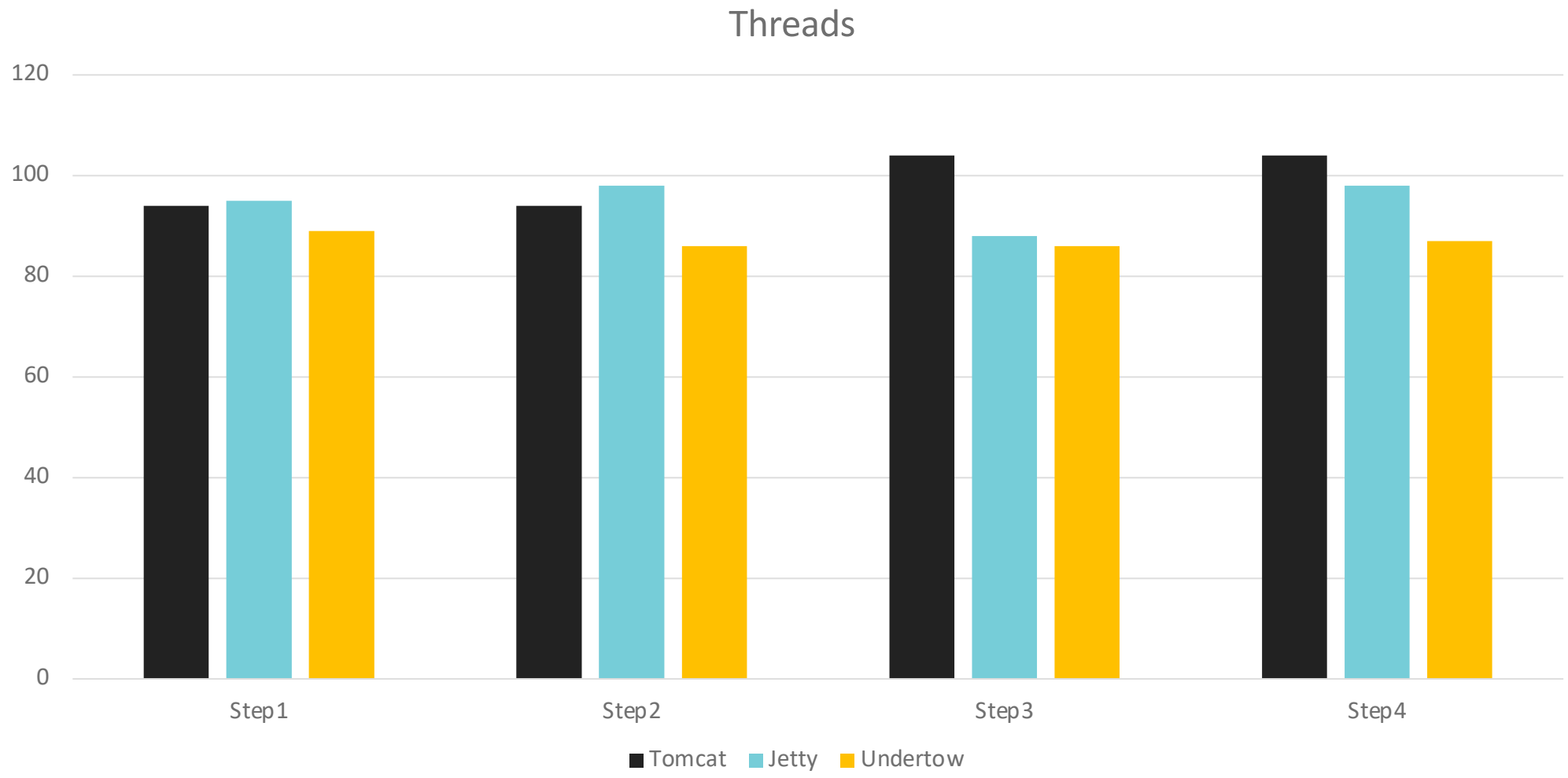
EMBEDDED SERVERS' COMPARISON



EMBEDDED SERVERS' COMPARISON



EMBEDDED SERVERS' COMPARISON



WHERE IS MY BOOT-UP TIME, LEBOWSKI?

Topics:

- BOOT-UP TIME
- IN UNDERTOW WE TRUST

BOOT-UP TIME

Now we will take a look at boot-up metrics on different embedded web servers.
Due to the fact, that we are using fewer libraries than the default HFJS starter it decreases the boot time up to 2 times.

Tomcat	Jetty	Undertow
~ 55,5s	~46,5s	~ 45,5s

Moreover, to decrease the boot-up time we've tried to add spring-indexer. Nevertheless, the spring-indexer is not compatible with hapi-fhir. In spite of this, I would like to share the benefits it could bring to your projects. While classpath scanning is very fast, it is possible to improve the startup performance of large applications by creating a static list of candidates at compilation time. In this mode, all modules that are targets of component scanning must use this mechanism. It creates META-INF/spring.components were all the beans described.
Actually, it increases the build time in few seconds, but every cloud has a silver lining.

IN UNDERTOW WE TRUST

Due to the fact of most stable results of:

- Transactions Per Working Day
- Response Time
- Resources Consumption
- Boot up time

We chose the Undertow as the main embedded server for HAPI-FHIR server.

I CAN'T BE WORRYING ABOUT THAT. LIFE GOES ON, MAN.

Topics:

- COMPARING JDK 11 AND JDK 17
- COMPARING G1, ZGC

COMPARING JDK 11 AND JDK 17

After all the tests we've already described we are going to use Undertow as the embedded server.
Now we are going to compare the performance using JDK 11 and JDK 17.
The result are below.

JDK	TRANSACTIONS PER WORKING DAY	TRANSACTIONS PER SECOND	RESPONSE TIMES OVER TIME	HITS PER SECOND	NODE CPU	NODE RAM (GiB)	JVM RATE (MAX ops/s)	JVM HEAP (GiB)	JVM NON-HEAP (Mib)	JVM CPU (%)	THREADS (MAX)	HIKARI POOL (MAX)	HIKARI CONNECTIONS TIME (MAX ms)
11	460800	16	~80	~40	~0,380	~4,65	~31	~2,57	~251	~28	~87 (27 runnable, 26 waiting, 40 timed-waiting)	7 active, 16 idle	69 usage, 51 creation
17	576000	20	~80	~54	~0,320	~3,83	~31	~2,24	~242	~17	~86 (28 runnable, 26 waiting, 41 timed-waiting)	6 active, 16 idle	110 usage, 77 creation

COMPARING G1, ZGC

After comparing the results of JDK 11 and JDK17 we are decided to choose the latest one LTS version (JDK 17).
Let's check the performance for ZGC as well.

GC TYPE	TRANSACTIONS PER WORKING DAY	TRANSACTIONS PER SECOND	RESPONSE TIMES OVER TIME	HITS PER SECOND	NODE CPU	NODE RAM (GiB)	JVM RATE (MAX ops/s)	JVM HEAP (GiB)	JVM NON-HEAP (Mib)	JVM CPU (%)	THREADS (MAX)	HIKARI POOL (MAX)	HIKARI CONNECTIONS TIME (MAX ms)	POSTGRES STATEMENTS CALLS (ops/s)
G1	576000	20	~100	~54	~0,320	~3,83	~31	~2,24	~242	~17	~86 (28 runnable, 26 waiting, 41 timed-waiting)	6 active, 16 idle	110 usage, 77 creation	max: 488, avg: 243
ZGC	460800	16	~80	~40	~0,350	~3,09	~32	~3,08	~219	~15	~86 (25 runnable, 27 waiting, 43 timed-waiting)	5 active, 16 idle	80 usage, 53 creation	max: 497, avg: 248

HA HEY, THIS IS A PRIVATE RESIDENCE MAN.

Topics:

- NEXT STEPS
- FINAL SETUP
- SUMMARY

NEXT STEPS

I. Migration from Postgres 13.4 to Postgres 14.1 due to having a lot of performance tunings, such as:

- Numerous performance improvements have been made for parallel queries, heavily-concurrent workloads, partitioned tables, logical replication, and vacuuming;
- B-tree index updates are managed more efficiently, reducing index bloat;
- VACUUM automatically becomes more aggressive, and skips inessential clean-up, if the database starts to approach a transaction ID wraparound condition;
- Ensure that parallel VACUUM doesn't miss any indexes;
- Fix REINDEX CONCURRENTLY to preserve operator class parameters that were attached to the target index;
- Avoid O(N^2) behavior in some list-manipulation operations;
- Add more defensive checks around B-tree posting list splits.

II. Enabling caching

FINAL SETUP

Let’s compare the results of tuning.

TUNING	JDK	GC	SERVER	GRADLE	POSTGRES	POSRGRES VM RAM	POSRGRES VM CORE	POSTGRES VM SSD	JVM RAM	JVM CORE
BEFORE	11	G1	Tomcat	7.1	13.4	32 GiB	8	2 TB	16 GiB	4
AFTER	17	G1	Undertow	7.3	13.4	4 GiB	2	500 GB	3 GiB	2

In comparing with the basic setup, we have the next benefits:

1. Optimized our costs up to 4 times:
   1.1. Resources consumption on HAPI FHIR JPA SERVER decreased from 16GB RAM and 4 Cores to 3GB RAM and 2 Cores. As a result, we can deploy more instances for the same costs;
   1.2. PostgreSQL moved from 32GB RAM and 8 Cores to 4GiB RAM and 2 Cores.
2. Decreased boot-up time in 2 times;
3. Improved performance almost in 2 times.

The full description of tuning and the definition of final spring configuration you can find there. Star it!

Thank you!

For more information, contact

vadzim_kavalkou@epam.com

vadzim.kavalkou@gmail.com

https://www.linkedin.com/in/vadzimkavalkou/

https://github.com/fragaLY