

Datasource hikaricp

Hikaricp es una implementación del tipo jdbc datasource y una opción bastante interesante para la configuración de las conexiones a nivel de backend

Configuración del datasource por defecto spring boot 2.x

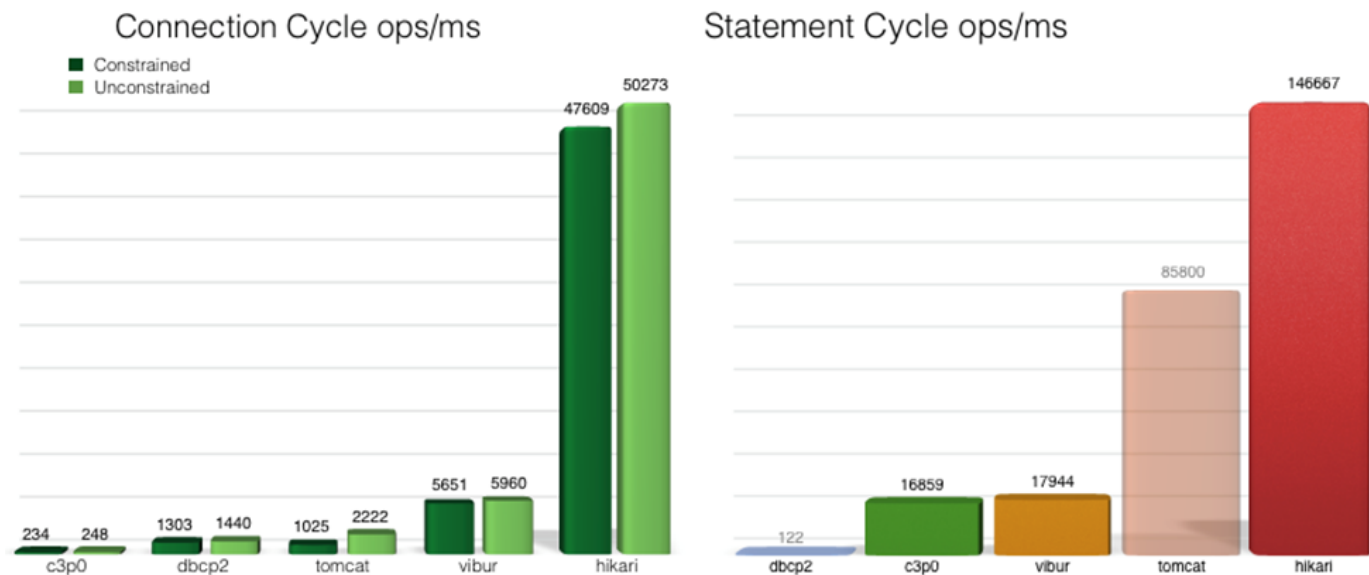
Como sabemos en el proyecto RSI se venía manejando una configuración estándar para el datasource que controla y administra las conexiones y el pool en nuestra aplicación, el cual fue remplazado por una implementación de wildfly la cual es mucho mas configurable y bastante robusta, con funciones de monitoreo y muy rica en opciones, sin embargo el tener una configuración que satisface nuestras necesidades no nos abole de conocer otras opciones que tenemos para configurar nuestro datasource por si bajo otras condiciones llega a ser necesario tener opciones tal vez mas practicas e igualmente funcionales y poder saber cómo abordar dichas situaciones y elegir lo mejor para cada caso.

En este documento se explicará la configuración hikari, como configurarla y sus alcances o límites, tanto desde el properties como por medio de una implementación.

HIKARI DATASOURCE



La implementación de hikari como jdbc datasource es una opción bastante interesante puesto que en su documentación oficial Promete ser ligera y aun así tener el mejor desempeño, en cuanto a lo ligera de la librería podemos confirmar que tiene un peso bastante mínimo de alrededor de 130kb y ofrece un montón de parámetros de configuración para solventar los distintos problemas que pueda llegar a presentar nuestra aplicación y ser lo más optima posible.



Según documentación esta implementación es la más rápida y obtiene muy buenos resultados y está construida muy inteligente y eficiente.

Configuración bajo propiedades en archivo properties

Para configurar el datasource en spring boot 2.x podemos hacerlo con la configuración por defecto es decir únicamente mediante el archivo `application.properties` al indicar algunas de las variables modificables y con esto tener un datasource funcional, ahora bien, si nosotros queremos tener un control mucho mayor sobre nuestro datasource o necesitamos un mayor nivel de personalización basta con crear una implementación propia de `DataSource` y esto se hace con una clase `@configuration` y un hikari datasource.

En Spring boot están las propiedades `Spring.DataSource.hikari` las cuales toma para la configuración del hikari por defecto haciendo uso de la clase `HikariConfig` y es necesario incluir las clásicas **`Spring.datasource.url`** , **`Spring.datasource.username`** , **`Spring.datasource.password`** pues estas propiedades son especiales y globales para cualquier tipo de implementación y hikari no es la excepción y las toma de ellas.

Una configuración posible por medio de únicamente definición de propiedades en el archivo `application.properties` es la siguiente:

```

1  #tiempo de espera en una conexión
2  spring.datasource.hikari.connection-timeout = 20000
3  #maximum pool size
4  spring.datasource.hikari.maximum-pool-size= 3
5  #maximum idle time for connection
6  spring.datasource.hikari.idle-timeout=60000
7  # maximum lifetime in milliseconds of a connection in the pool after it is clo
8  spring.datasource.hikari.max-lifetime=60000
9  #default auto-commit behavior.
10 spring.datasource.hikari.auto-commit =true

```

```
11 | #Datasource pool name
12 | spring.datasource.hikari.pool-name=HikariPool
```

“

En resumen, con esto tenemos una configuración administrada automáticamente por spring y no debemos preocuparnos por nada, si vemos la prueba de carga con 4000 usuarios simultáneos podemos ver como tiene un rendimiento bastante bueno.

```
Results of period #1 (from 3 sec to 21 sec ):
*****
Completed Clicks: 2040 with 0 Errors (=0,00%)
Average Click Time for 4.000 Users: 2.239 ms
Successful clicks per Second: 115,02 (equals 414.084,33 Clicks per Hour)

Results of period #2 (from 21 sec to 33 sec ):
*****
Completed Clicks: 1198 with 0 Errors (=0,00%)
Average Click Time for 4.000 Users: 1.083 ms
Successful clicks per Second: 95,42 (equals 343.513,33 Clicks per Hour)

Results of period #3 (from 33 sec to 47 sec ):
*****
Completed Clicks: 762 with 0 Errors (=0,00%)
Average Click Time for 4.000 Users: 1.335 ms
Successful clicks per Second: 54,28 (equals 195.421,29 Clicks per Hour)

Results of complete test
*****

** Results per URL for complete test **

URL #1 (tes1): Average Click Time 1.721 ms, 4.000 Clicks, 0 Errors

Total Number of Clicks: 4.000 (0 Errors)
Average Click Time of all URLs: 1.721 ms
```

En este ejemplo vemos como con un datasource hikari para 4000 registros de obtiene una muy buena métrica, aunque el web stres no es la mejor herramienta y los resultados dependen de otros factores como red y demás igualmente es bastante rápido y eficiente.

“

Sin embargo, esta implementación posee una limitación con las transacciones pues no incluye un manejador de transacciones, en dado caso es necesario crear una implementación propia en una clase @configuration y utilizar la propiedad @EnableTransactionManagement, estas dos anotaciones en conjunto nos permiten declarar tanto el Bean del datasource de hikari como también el platformTransacionManager para el manejo de las transacciones.

¿Ahora bien, por qué es necesario tener un manejador de transacciones?

Bueno sucede que por defecto al desplegar en un servidor nuestra aplicación las transacciones no son manejadas de la manera esperada, me explico, la manera en la que funcionan las conexiones o sesiones y como se realizan los commits a la base de datos por defecto no es la deseada para el proyecto RSI, donde se lleva a cabo una auditoria del lado del servidor backend y en la cual se necesitan un identificador único o un único commit por transacción que se realice...

La manera en la que spring lo hace por defecto es enlazando un identificador o commit separado a cada operación que realiza y no a toda la transacción, esto lógicamente es solventado con la configuración mencionada sin embargo vamos a profundizar un poco en como se manejan las transacciones y la importancia de esto.

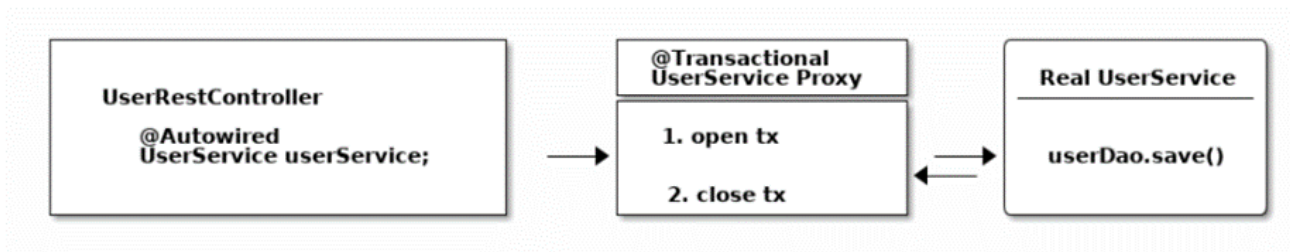
```
1 public class UserService {
2     public void registerUser(User user){
3         var connection = dataSource.getConnection(); // (1)
4         try(connection){
5             connection.setAutoCommit(false); // (1)
6             //...validate the user
7             userDao.save(user); // (2)
8             connection.commit(); // (1)
9         } catch (SQLException e){
10             connection.rollback(); // (1)
11         }
12     }
13 }
```

Cuando nosotros entramos en una transacción sucede básicamente lo descrito en la imagen primero se solicita una conexión al datasource, luego por medio de esta conexión se deshabilita el autocommit, se realizan los save, update, delete requeridos y luego se realiza el commit manual para garantizar que solo haya un commit por transacción, en caso de error se produce el rollback y la conexión finaliza y regresa al pool.

Es obvio que nosotros no trabajamos este comportamiento manual, de hecho, por defecto se maneja autocommit = true, **entonces como lo hace spring?**

@Transactional en spring boot

Cuando nosotros inyectamos un servicio en un controller este se inyecta en 2 partes, una parte actúa de proxy y la otra si es la implementación, la parte que funciona como proxy se encarga de realizar los pasos en caso de que haya una transacción y maneja el comportamiento automático por nosotros, pero es necesario tener un manejador de transacciones para obtener el comportamiento deseado de otra forma no se logra y la auditoria se ve comprometida.



“

La documentación proporcionada en internet nos dice que cuando un Bean es anotado con `@transactional` o un método, (entiéndase Bean como cualquier objeto que se agrega al contenedor de objetos de spring y sobre los cuales tiene control en tiempo de ejecución) y se maneja el concepto de proxy donde el proxy es esa capa intermedia que se encarga de hacer el proceso de commit mencionado previamente.

Ahora para lograr este comportamiento el proxy necesita un datasource para abrir y cerrar conexiones o transacciones automáticamente, para el rollback etc. Y spring boot maneja una interfaz que se encarga de administrar todo ese manejo de transacciones y se llama **PlatformTransactionManager**.

Manejador de transacciones

Hay muchas implementaciones que nos sirve y la mas simple y que parcialmente nos funciona es una llamada **DataSourceTransactionManager** y se construye en base a un datasource, el único problema de esta implementación es que se maneja por separado a la capa de hibernate de modo que si tenemos un método transactional y lo manejamos con la clase **DataSourceTransactionManager** entonces Spring creara y cerrara conexiones en el datasource automáticamente pero si nuestro código llama a hibernate como es el caso del proyecto RSI donde jpa e hibernate actúan como orm pues el mismo llama a su propio SessionFactory para crear y cerrar nuevas sesiones y administrar su estado sin spring y nuestra configuración.

En una prueba de carga donde se presenta esta situación tan pronto sobrepasan los usuarios simultáneos a la cantidad de conexiones disponibles en el pool el servicio queda inhabilitado y se presume es por este comportamiento, como solución en foros y en páginas se recomienda utilizar un **HibernateTransactionManager** o **JpaTransactionManager**, y esta manera es similar a como se trabaja la implementación en informix.

Análisis de alternativas entre **HibernateTransactionManager** or **JpaTransactionManager**.

“

Según documentación encontrada en foros el utilizar Jpa en vez de hibernate requiere una parte adicional y es una configuración para el entity manager que es una interface que maneja las entidades y ya no es necesario ninguna dependencia de hibernate sobre las mismas, es una practica bastante usada por desarrolladores, por el contrario el uso de hibernate haría necesaria una dependencia de las clases y sería necesario un sessionFactory que tuviera en cuenta el datasource para trabajar en conjunto pero todo administrado desde hibernate.

Ahora bien, el usar JpaTransactionManager no hará que no se dependa de hibernate ya que JPA e hibernate trabajan juntos, jpa como especificación e hibernate como implementación, es quien provee las funcionalidades que describe el jpa, sin embargo el hacer que todo esté mapeado sobre jpa facilita el aislamiento e incluso reduce tiempo si se quiere cambiar de proveedor de jpa mas adelante, y pues jpa también es preferido por sus interfaces mas estables y configuración mas amigable.

“

Por conclusión se adopta la configuración jpa.

Análisis de test de carga con configuración JPA para el datasource HIKARI

**** Test Logfile by Webserver Stress Tool 8.0.0.1010 Enterprise Edition (Freeware) ****© 1998-2012 Paessler AG, <http://www.paessler.com>

Test run on 11/03/2022 2:43:02 p. m.

**** Project and Scenario Comments, Operator ******Results of period #1 (from 3 sec to 20 sec):**

Completed Clicks: 1707 with 0 Errors (=0,00%)

Average Click Time for 4.000 Users: 2.995 ms

Successful clicks per Second: 103,27 (equals 371.775,04 Clicks per Hour)

Results of period #2 (from 20 sec to 33 sec):

Completed Clicks: 1384 with 0 Errors (=0,00%)

Average Click Time for 4.000 Users: 2.193 ms

Successful clicks per Second: 109,02 (equals 392.454,43 Clicks per Hour)

Results of period #3 (from 33 sec to 47 sec):

Completed Clicks: 909 with 0 Errors (=0,00%)

Average Click Time for 4.000 Users: 864 ms

Successful clicks per Second: 61,52 (equals 221.456,19 Clicks per Hour)

Results of complete test

**** Results per URL for complete test ****

URL #1 (tes1): Average Click Time 2.233 ms, 4.000 Clicks, 0 Errors

Total Number of Clicks: 4.000 (0 Errors)

Average Click Time of all URLs: 2.233 ms

“

La configuración a nivel de spring boot es la siguiente, mediante una clase de configuración se habilita el manejo de transacciones e inmediatamente se crea el hikari datasource que nos interesa.

```

1  @Configuration
2  @EnableTransactionManagement
3  public class OracleDataSourceConfig {
4
5      @Autowired
6      private Environment env;
7
8      @Bean(name = "HikariDataSource")
9      public HikariDataSource hikariDataSource() {
10         final var ds = new HikariDataSource();

```

```

11         ds.setDataSourceClassName(env.getProperty("spring.datasource.data-sourceClassName"));
12         ds.addDataSourceProperty("url", env.getProperty("spring.datasource.url"));
13         ds.addDataSourceProperty("user", env.getProperty("spring.datasource.username"));
14         ds.addDataSourceProperty("password", env.getProperty("spring.datasource.password"));
15         ds.setPoolName(env.getProperty("spring.datasource.pool-name"));
16         return ds;
17     }
18
19     @Bean(name = "entityManagerFactory")
20     public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
21         LocalContainerEntityManagerFactoryBean entityManager = new LocalContainerEntityManagerFactoryBean();
22         entityManager.setDataSource(hikariDataSource());
23         entityManager.setPackagesToScan("co.edu.uis.rh.admonpersonal.model");
24
25         var vendorAdapter = new HibernateJpaVendorAdapter();
26         entityManager.setJpaVendorAdapter(vendorAdapter);
27
28         Map<String, Object> hibernateProperties = new HashMap<>();
29         hibernateProperties.put("hibernate.hbm2ddl.auto", env.getProperty("spring.datasource.hibernate.hbm2ddl.auto"));
30         hibernateProperties.put("hibernate.show_sql", env.getProperty("spring.datasource.hibernate.show_sql"));
31         hibernateProperties.put("hibernate.format_sql",
32             env.getProperty("spring.jpa.properties.hibernate.format_sql"));
33         entityManager.setJpaPropertyMap(hibernateProperties);
34         return entityManager;
35     }
36
37
38     @Bean(name = "transactionManager")
39     public PlatformTransactionManager transactionManagerFactory() {
40         var transactionManager = new JpaTransactionManager();
41         transactionManager.setEntityManagerFactory(entityManagerFactory().getEntityManager());
42         return transactionManager;
43     }
44
45
46
47 }

```

Como lo mencionamos anteriormente una configuración por si sola del datasource no producirá los resultados deseados del manejo de transacciones y es necesario para ello incluir el entity manager el cual trabajará de la mano con el datasource para el manejo de las conexiones y ya no se hará por separado ni se presentará el conflicto con hiberanate y el datasource.

Y tambien necesitamos como se mencionó anteriormente un PlatformTransactionManager para el manejo de las transaccionalidad del proyecto, estos pasos son necesarios porque con el entity manager garantizamos

que se están tomando las conexiones del datasource y no por separado(lo que genera una fuerte deficiencia en el rendimiento de nuestra aplicación) y a su vez el manejador de transacciones basado en JPA que nos da un manejo mucho mas agradable y comodo de las entidades.

¿Cómo se maneja desde el proyecto de integración y cómo podría adaptarse a los demás proyectos?

La única diferencia está en el datasource, de resto la configuración es la misma, y pues el hecho de cambiar el datasource requiere unos pequeños cambios en la clase, vamos a realizarlos y a mirar el test de carga para ver si dicha implementación también funciona y solventa nuestros inconvenientes.

```
1 | @Bean(name = "dataSource")
2 |     public DataSource oracleDataSource() {
3 |         var dataSource = new DriverManagerDataSource();
4 |         dataSource.setUrl(env.getProperty("spring.datasource.url"));
5 |         dataSource.setUsername(env.getProperty("spring.datasource.username"));
6 |         dataSource.setPassword(env.getProperty("spring.datasource.password"));
7 |         return dataSource;
8 |     }
```



La prueba de carga nos arroja resultados bastante menos eficientes en comparación con una configuración de datasource en vez de driver puesto que el driver maneja una conexión y el datasource es mucho mas flexible y configurable y nos define varias conexiones.

Results of period #10 (from 94 sec to 104 sec):

Completed Clicks: 62 with 0 Errors (=0,00%)

Average Click Time for 1.000 Users: 93.720 ms

Successful clicks per Second: 5,98 (equals 21.522,52 Clicks per Hour)

Results of period #11 (from 104 sec to 115 sec):

Completed Clicks: 99 with 0 Errors (=0,00%)

Average Click Time for 1.000 Users: 105.352 ms

Successful clicks per Second: 9,54 (equals 34.348,96 Clicks per Hour)

Results of period #12 (from 115 sec to 125 sec):

Completed Clicks: 102 with 34 Errors (=33,33%)

Average Click Time for 1.000 Users: 114.390 ms

Successful clicks per Second: 6,59 (equals 23.708,01 Clicks per Hour)

Results of complete test

**** Results per URL for complete test ****

URL #1 (tes1): Average Click Time 72.476 ms, 682 Clicks, 34 Errors

Total Number of Clicks: 682 (34 Errors)

Average Click Time of all URLs: 68.863 ms



En el proyecto de integración sucede lo mismo y los resultados no son los óptimos.

Average Click Time for 1.000 Users: 83.785 ms
 Successful clicks per Second: 8,01 (equals 28.830,83 Clicks per Hour)

Results of period #10 (from 93 sec to 103 sec):

Completed Clicks: 65 with 0 Errors (=0,00%)
 Average Click Time for 1.000 Users: 95.140 ms
 Successful clicks per Second: 6,30 (equals 22.674,95 Clicks per Hour)

Results of period #11 (from 103 sec to 114 sec):

Completed Clicks: 70 with 0 Errors (=0,00%)
 Average Click Time for 1.000 Users: 104.067 ms
 Successful clicks per Second: 6,75 (equals 24.299,73 Clicks per Hour)

Results of period #12 (from 114 sec to 124 sec):

Completed Clicks: 62 with 0 Errors (=0,00%)
 Average Click Time for 1.000 Users: 115.561 ms
 Successful clicks per Second: 5,98 (equals 21.538,74 Clicks per Hour)

Results of complete test

** Results per URL for complete test **

URL #1 (tes1): Average Click Time 74.032 ms, 614 Clicks, 0 Errors

Total Number of Clicks: 614 (0 Errors)
 Average Click Time of all URLs: 74.032 ms

11 Glossary

Conclusiones – recomendaciones.

“

La auditoria está funcionando y cualquiera de las configuraciones nos permite manejarla, sin embargo la configuración mas monitoreable y la que se usa actualmente (wildfly + jndi) es la que mejores prestaciones nos da, asegurando un buen rendimiento, asegurando un buen monitoreo, una flexibilidad y muchas opciones de configuración, por lo cual sugerimos mantenerla y en base al monitoreo y el comportamiento en los servidores hacer los ajustes para alcanzar el estado óptimo de la misma.

En cuanto a autowired vs persistence context para el manejador de entidades @EntityManager, se sugiere un persistence context puesto que nos permite definir varios de ellos como se maneja en integración, pero en el caso de los demás proyectos no presenta inconvenientes el utilizar @Autowired.

El driver de conexión puede funcionar bien bajo poca demanda según lo indicado en las pruebas de carga, en integración al ser un proyecto poco consumido parece ser una opción válida, pero en otros proyectos sería bueno discutirlo y tenerlo presente por si se adapta mejor la configuración como datasource en vez de como driver.

La configuración base de hikariCP es muy optima y habrá que explorar otras opciones de manejo de transacciones por si es necesario o más sencillo solo una configuración basada en properties que a su vez permita una auditoria completa (por ahora no se encontró).

Bibliografía

<https://github.com/brettwooldridge/HikariCP/> 

<https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing> 

<https://github.com/brettwooldridge/HikariCP/wiki/Spring-Hibernate-with-Annotations> 

<https://stackoverflow.com/questions/12337504/hibernatetransactionmanager-or-jpatransactionmanager> 

<https://www.baeldung.com/spring-boot-hikari> 

<https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth> 

<https://dzone.com/articles/spring-transaction-management-an-unconventional-gu> 