Почему мы решили переходить на R2DBC и чем это закончилось

Антон Котов, aalkotov@sberbank.ru – 15 июня 2022 года

SBER DEVICES

Agenda

1. What is R2DBC?
2. Why R2DBC?
3. Spring R2DBC
4. Portability of JDBC practices
5. R2DBC Cookbook

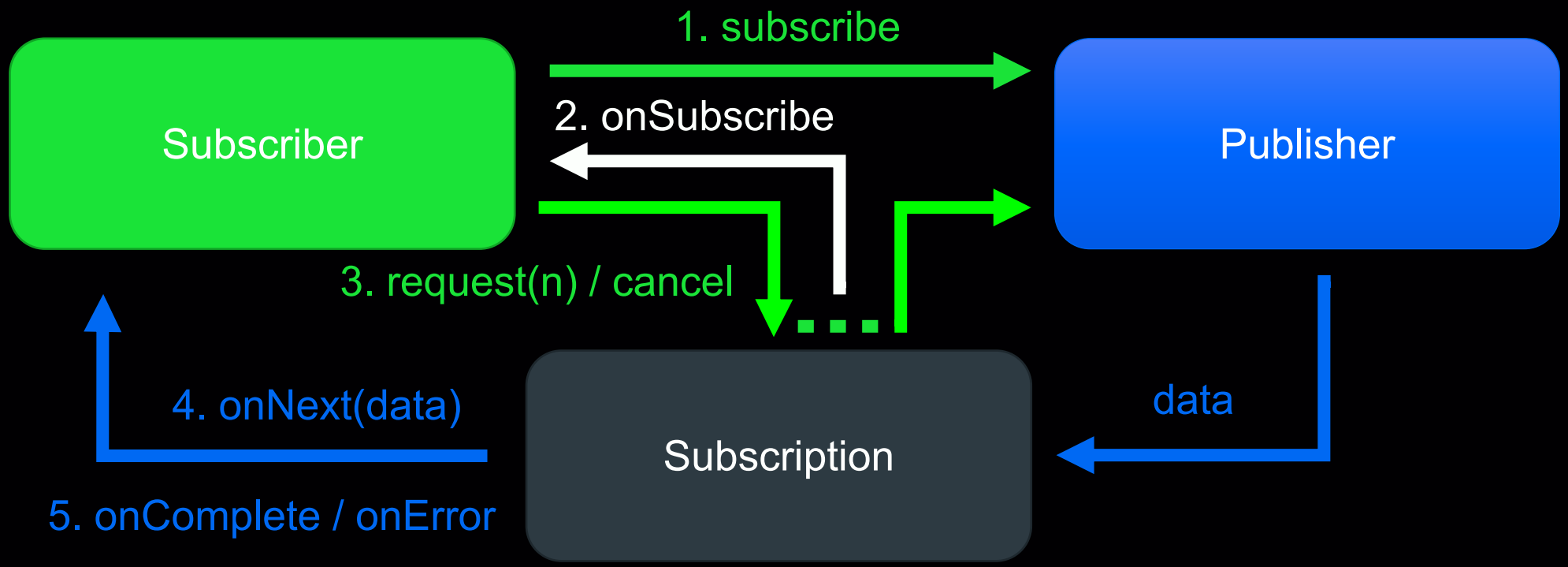
SBER DEVICES

What is R2DBC?

SBER DEVICES

What is R2DBC?

- The Reactive Relational Database Connectivity (R2DBC) project brings reactive programming APIs to relational databases.
- Based on the Reactive Streams specification.



SBER DEVICES

What is R2DBC? Connection

```
// JDBC
package java.sql;

public interface Connection {
  Statement createStatement();
  PreparedStatement
  prepareStatement(String sql);
  void commit();
  void rollback();
  void close();
  ...
}
```

```
// R2DBC
package io.r2dbc.spi;

public interface Connection {
  Batch createBatch();
  Statement
  createStatement(String sql);
  Publisher<Void> commitTransaction();
  Publisher<Void> rollbackTransaction();
  Publisher<Void> close();
  ...
}
```


SBER DEVICES

What is R2DBC? Statement

```
// JDBC
package java.sql;
public interface PreparedStatement {
  void setObject(...);
  void setNull(...);
  boolean execute();
  ResultSet getResultSet();
  int getUpdateCount();
  ResultSet getGeneratedKeys();
  ...
}
```

```
// R2DBC
package io.r2dbc.spi;
public interface Statement {
  Statement bind(...);
  Statement bindNull(...);
  Publisher<? extends Result>
    execute();
  Statement
    returnGeneratedValues(...);
  ...
}
```


SBER DEVICES

Project Reactor

Publisher

Mono

an Asynchronous

0-1 Result

Flux

an Asynchronous

Sequence of 0-N Items

SBER DEVICES

Example Query

```
ConnectionFactory connectionFactory = ConnectionFactories
   .get("r2dbc:h2:mem:///testdb");

Mono.from(connectionFactory.create())  // Mono<Connection>
   .flatMapMany(connection -> connection
     .createStatement("SELECT firstname FROM PERSON WHERE age > $1")
     .bind("$1", 42)
     .execute())  // Flux<Result>
   .flatMap(result -> result
     .map((row, rowMetadata) -> row.get("firstname", String.class)))
   .doOnNext(System.out::println)  // Flux<String>
   .subscribe();
```


SBER DEVICES

Good Example Query

```
Flux.usingWhen(
    connectionFactory.create(),
    connection -> connection
      .createStatement("SELECT firstname FROM PERSON WHERE age > $1")
      .bind("$1", 42)
      .execute(),
    connection -> connection.close()
  )  // Flux<Result>
  .flatMap(result -> result
    .map((row, rowMetadata) -> row.get("firstname", String.class)))
  .doOnNext(System.out::println)  // Flux<String>
  .subscribe();
```


SBER DEVICES

R2DBC Pool

```
// URL Connection Factory Discovery
// Creates a ConnectionPool wrapping an underlying ConnectionFactory
ConnectionFactory pooledConnectionFactory = ConnectionFactories.get(
  "r2dbc:pool:<driver>://<host>:<port>/<database>[?maxIdleTime=PT60S...]"
);

// Programmatic Configuration – Lifecycle support
ConnectionPoolConfiguration configuration =
    ConnectionPoolConfiguration.builder(connectionFactory)
  .postAllocate(connection ->
      Flux.from(connection.createStatement("SET schema = ...").execute())
    .flatMap(Result::getRowsUpdated).then())
  .preRelease(Connection::rollbackTransaction)
  .build();

ConnectionPool pool = new ConnectionPool(configuration);
```


SBER DEVICES

Why R2DBC?

SBER DEVICES

Why R2DBC?

Non-reactive approach

1. One connection – one thread.
2. Each tread consumes resources.
3. Blocking calls.
4. Context switching.
5. Difficult configuration.

Reactive approach

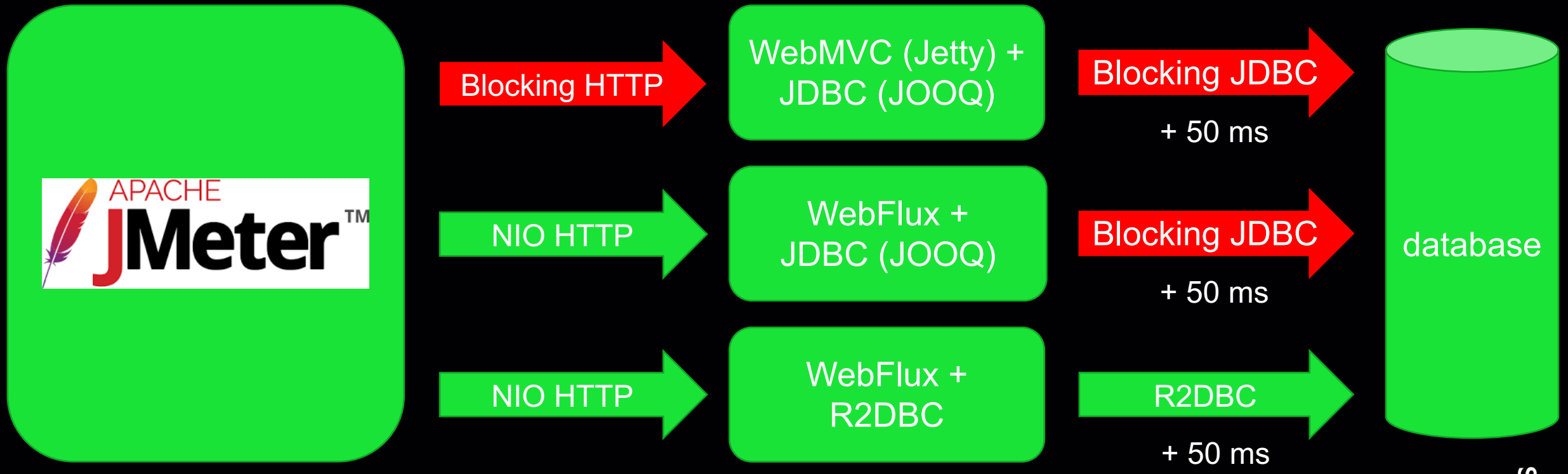
1. Relatively small thread group can serve multiple connections.
2. Optimal resource usage.
3. Non-blocking calls.
4. No or minimal context switching.
5. Scalability and resilience.

SBER DEVICES

Why R2DBC?

SBER DEVICES

Load test

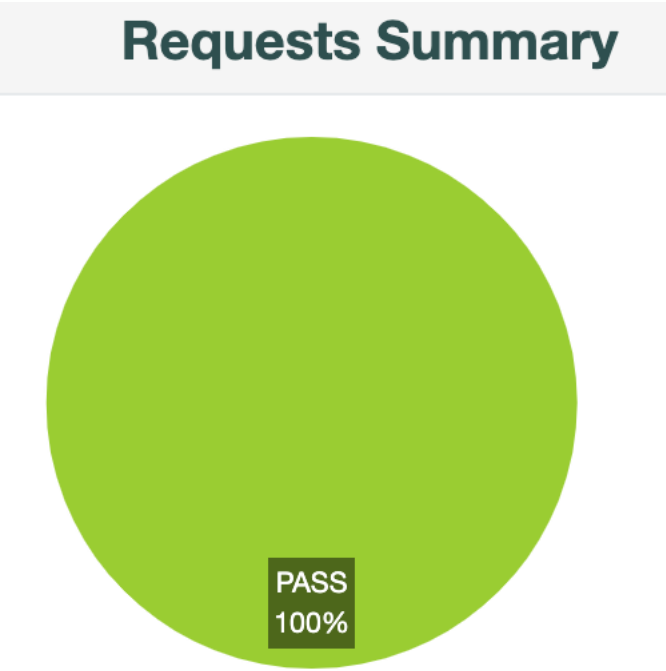


- 100 users, heavy requests – 10 operations for 200 ms each
- 5000 users, light request – one save operation in transaction

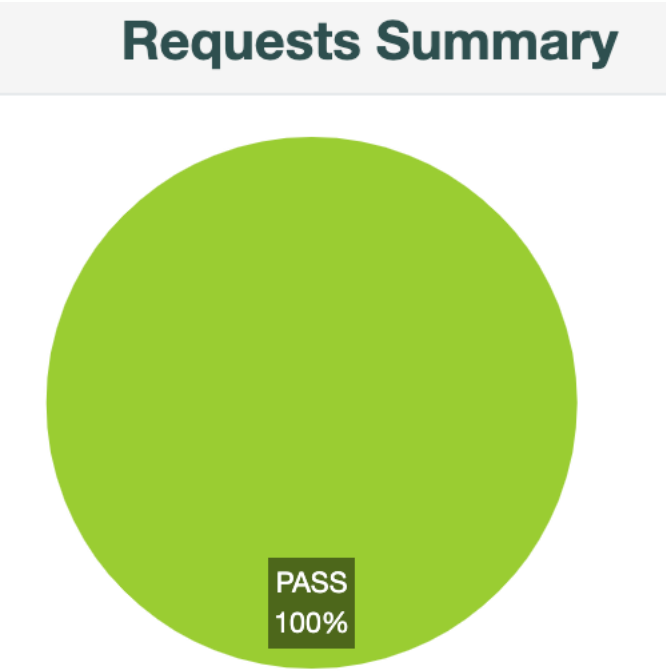
SBER DEVICES

Load test: 100 users, heavy requests

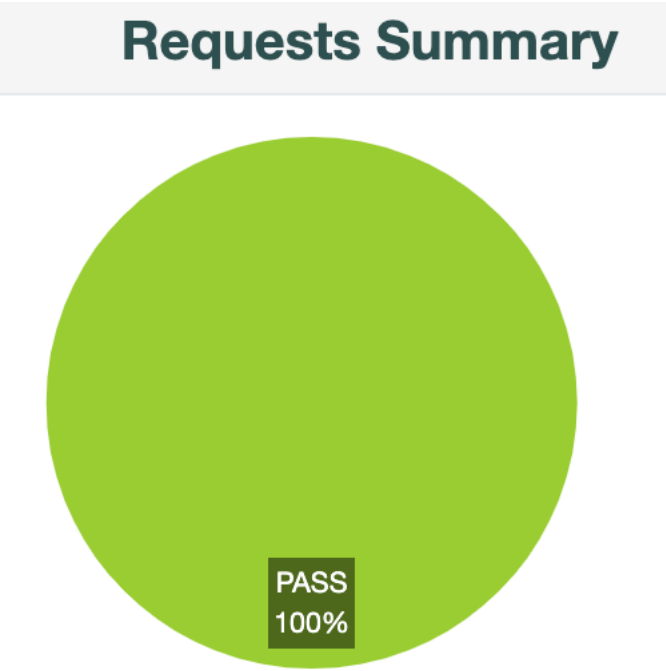
WebMVC + JDBC



WebFlux + JDBC

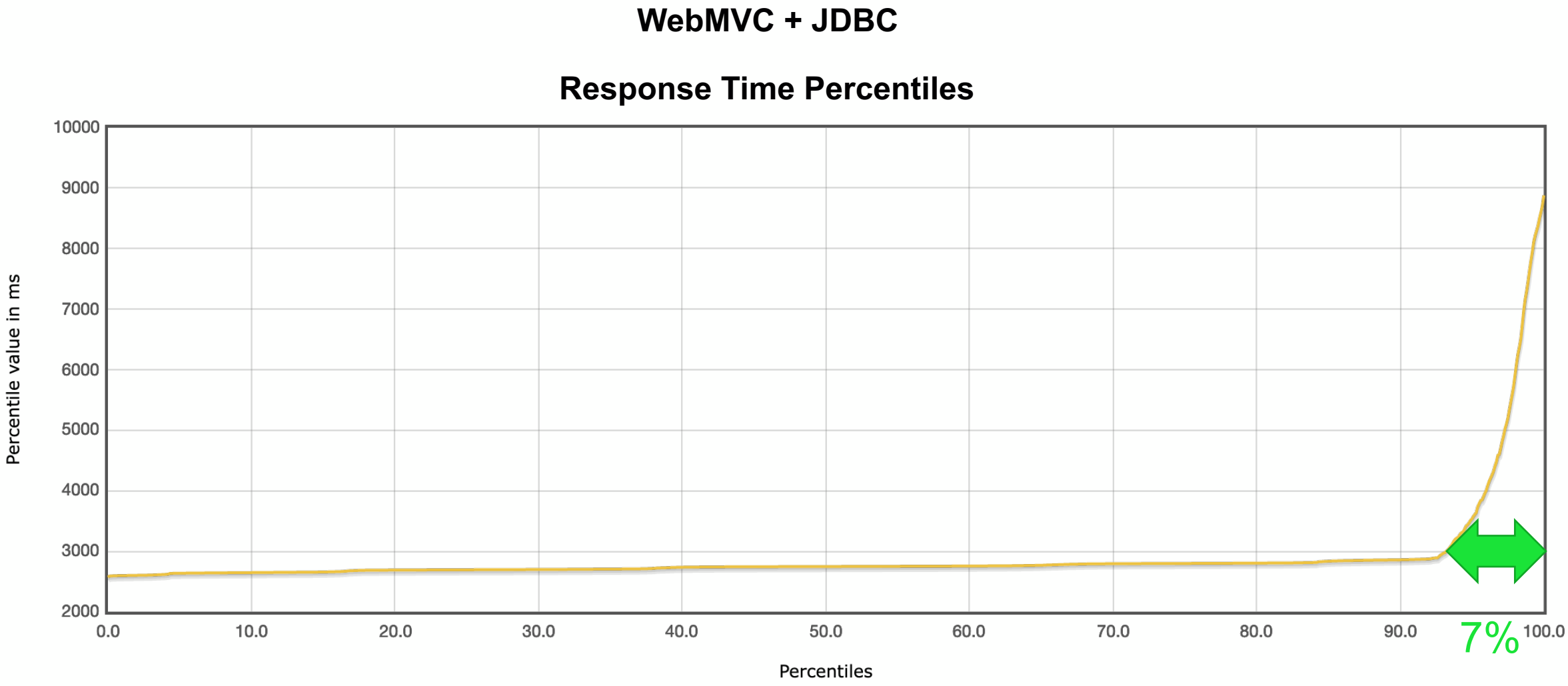


WebFlux + R2DBC



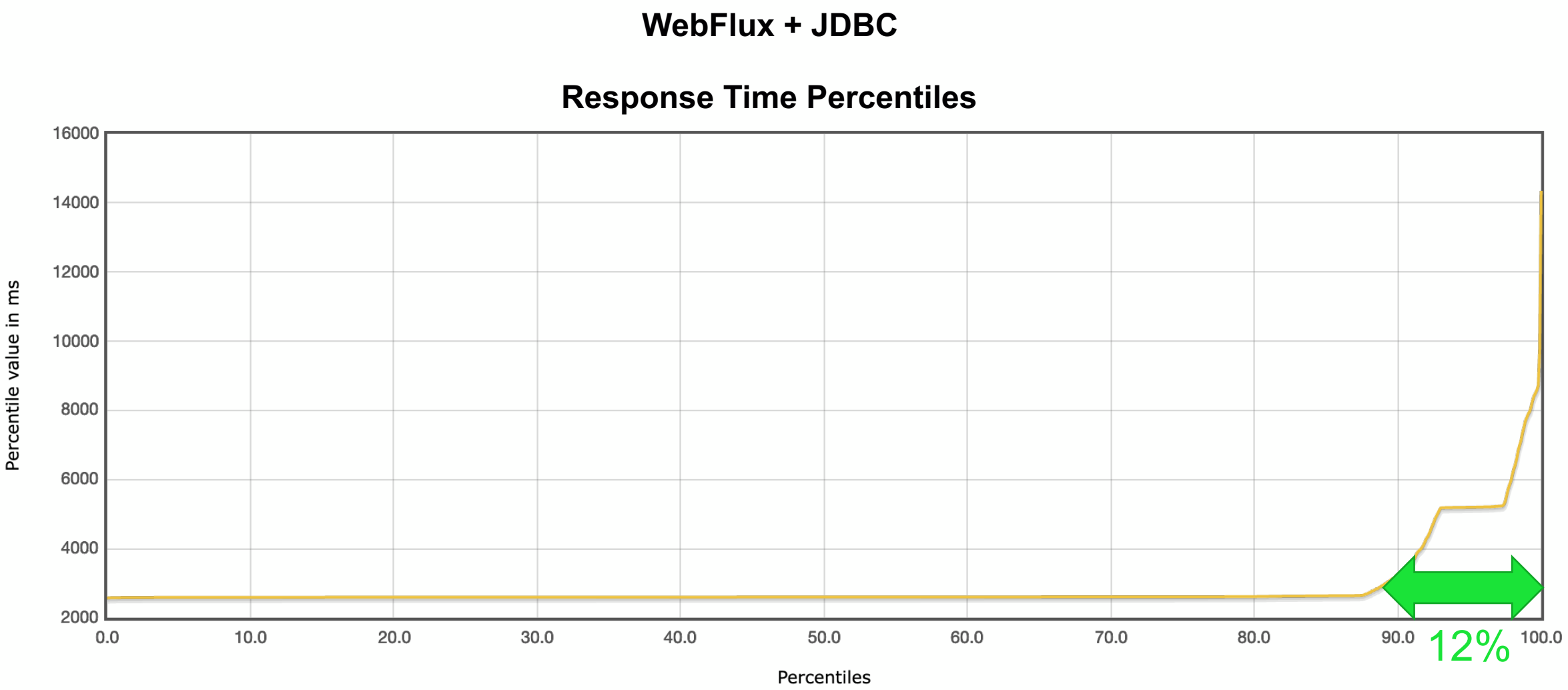
SBER DEVICES

Load test: 100 users, heavy requests



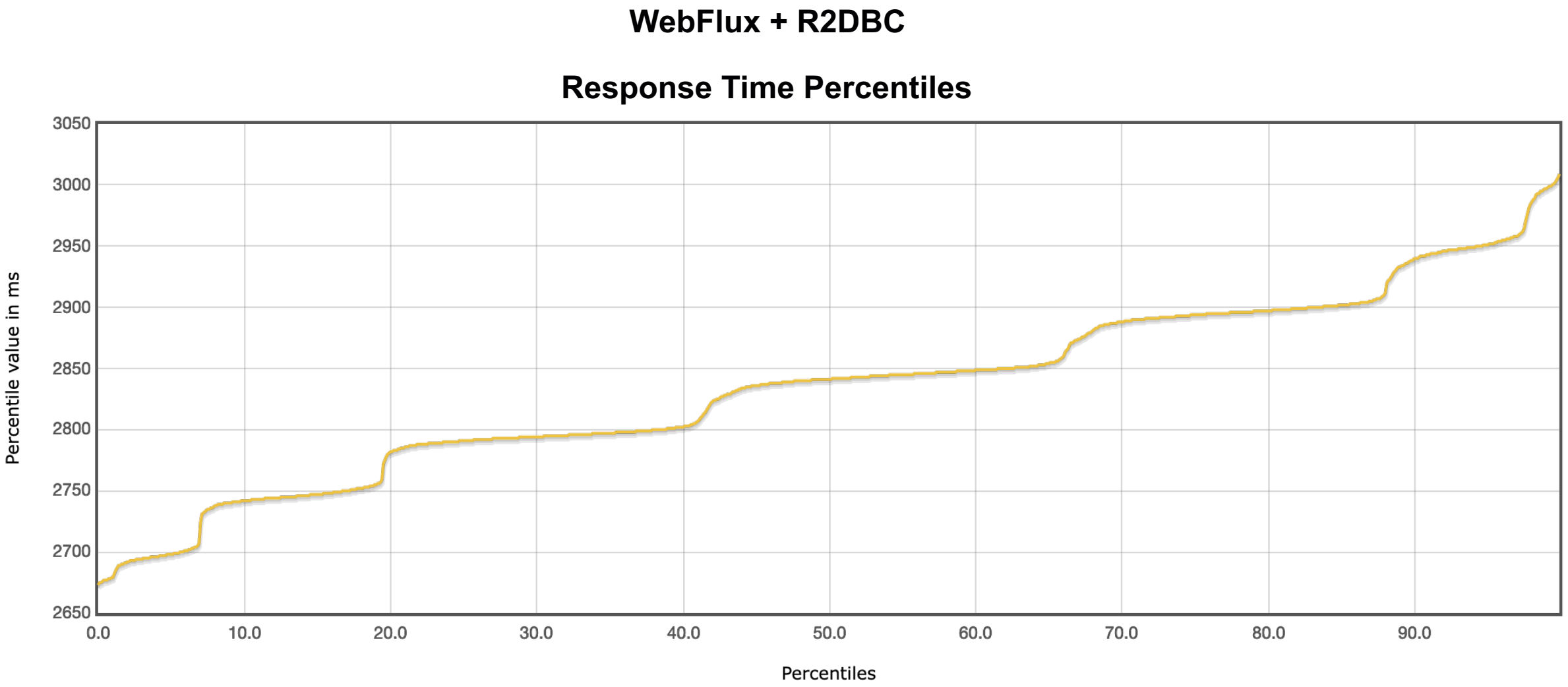
SBER DEVICES

Load test: 100 users, heavy requests



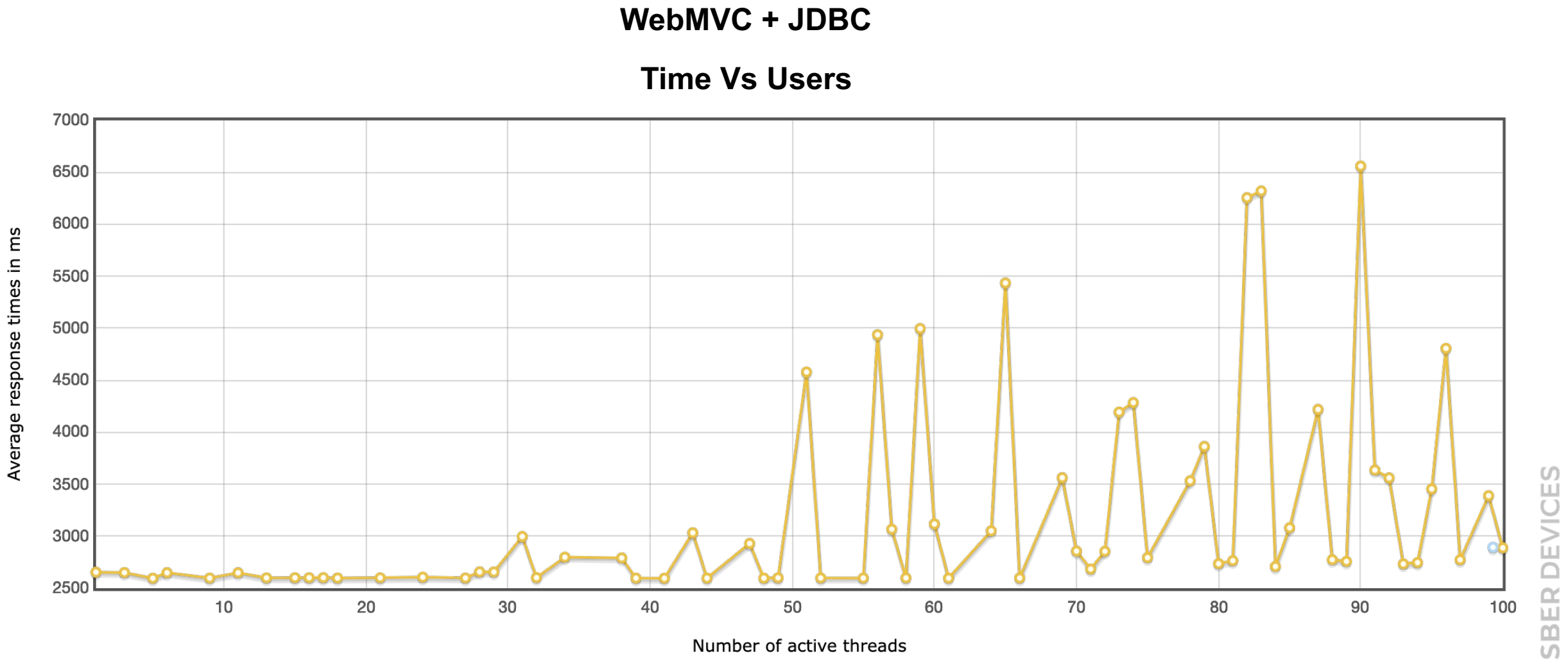
SBER DEVICES

Load test: 100 users, heavy requests



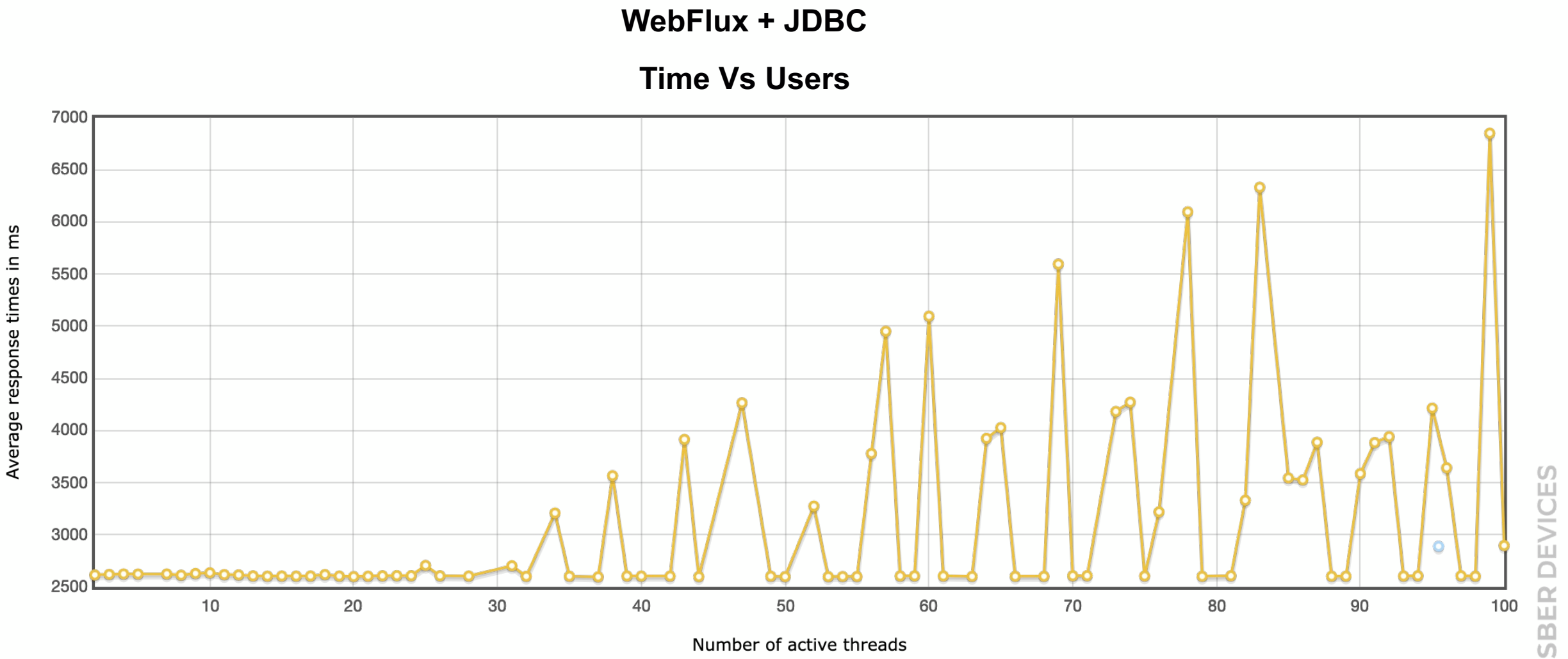
SBER DEVICES

Load test: 100 users, heavy requests



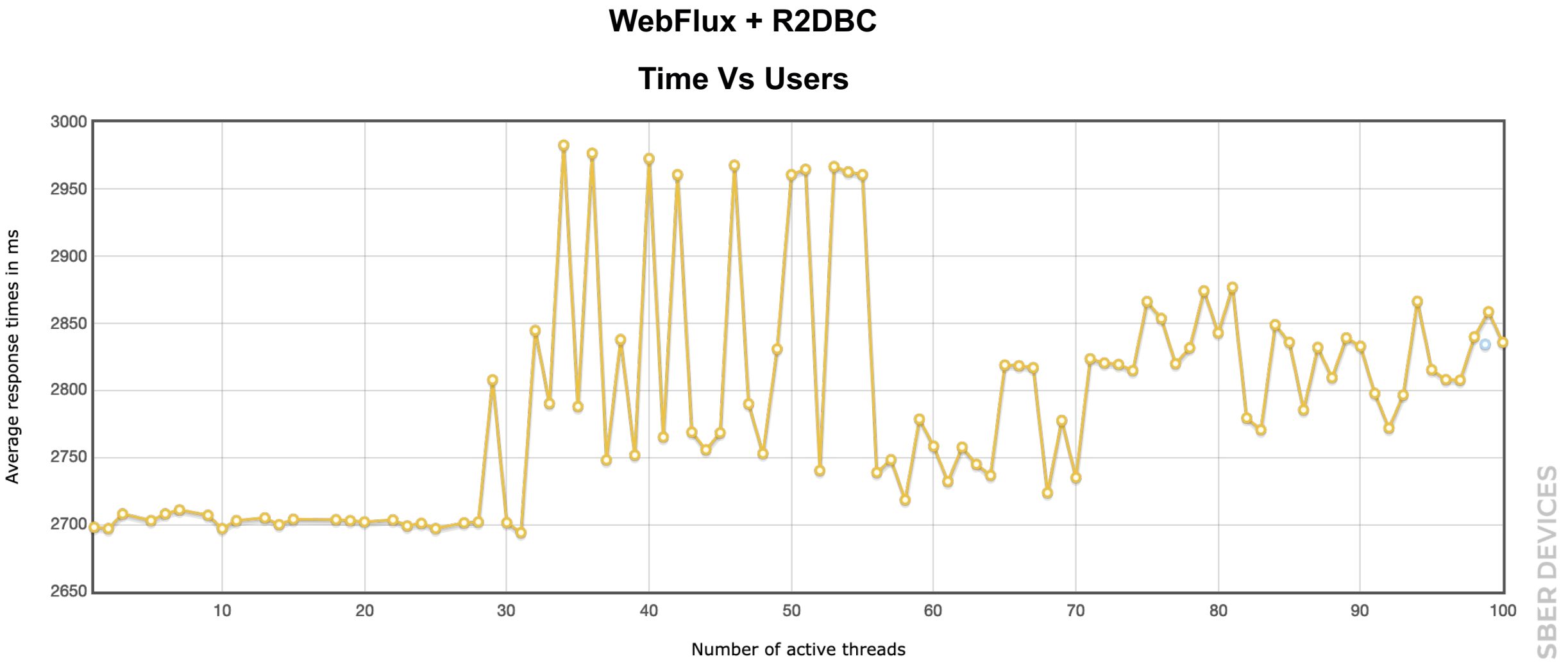
SBER DEVICES

Load test: 100 users, heavy requests



SBER DEVICES

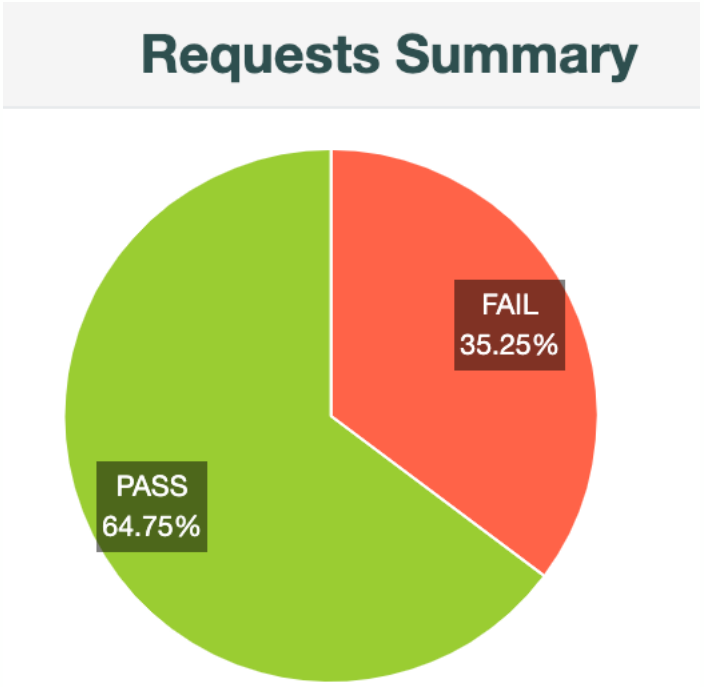
Load test: 100 users, heavy requests



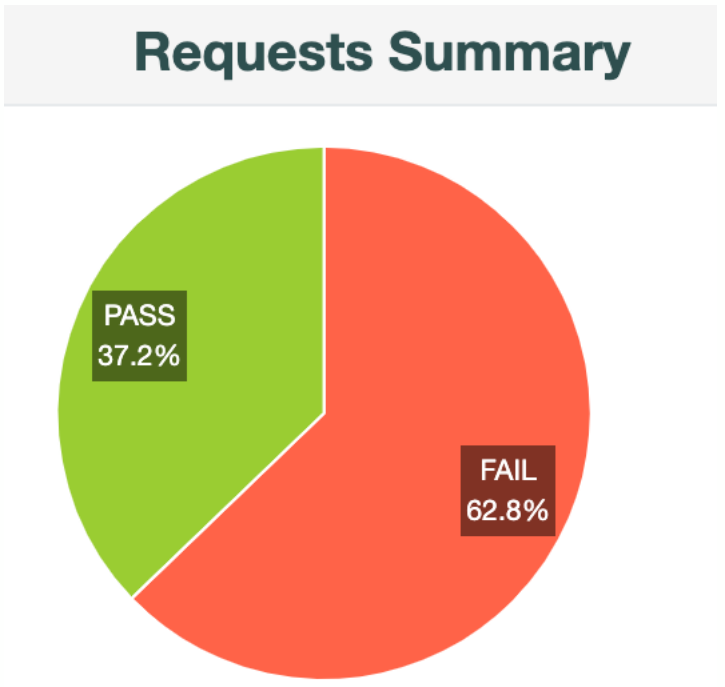
SBER DEVICES

Load test: 5000 users, light requests

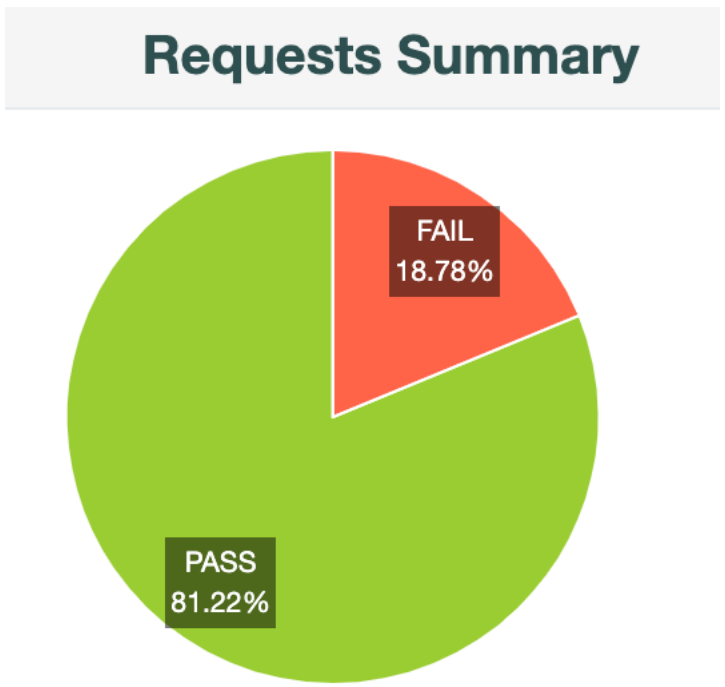
WebMVC + JDBC



WebFlux + JDBC

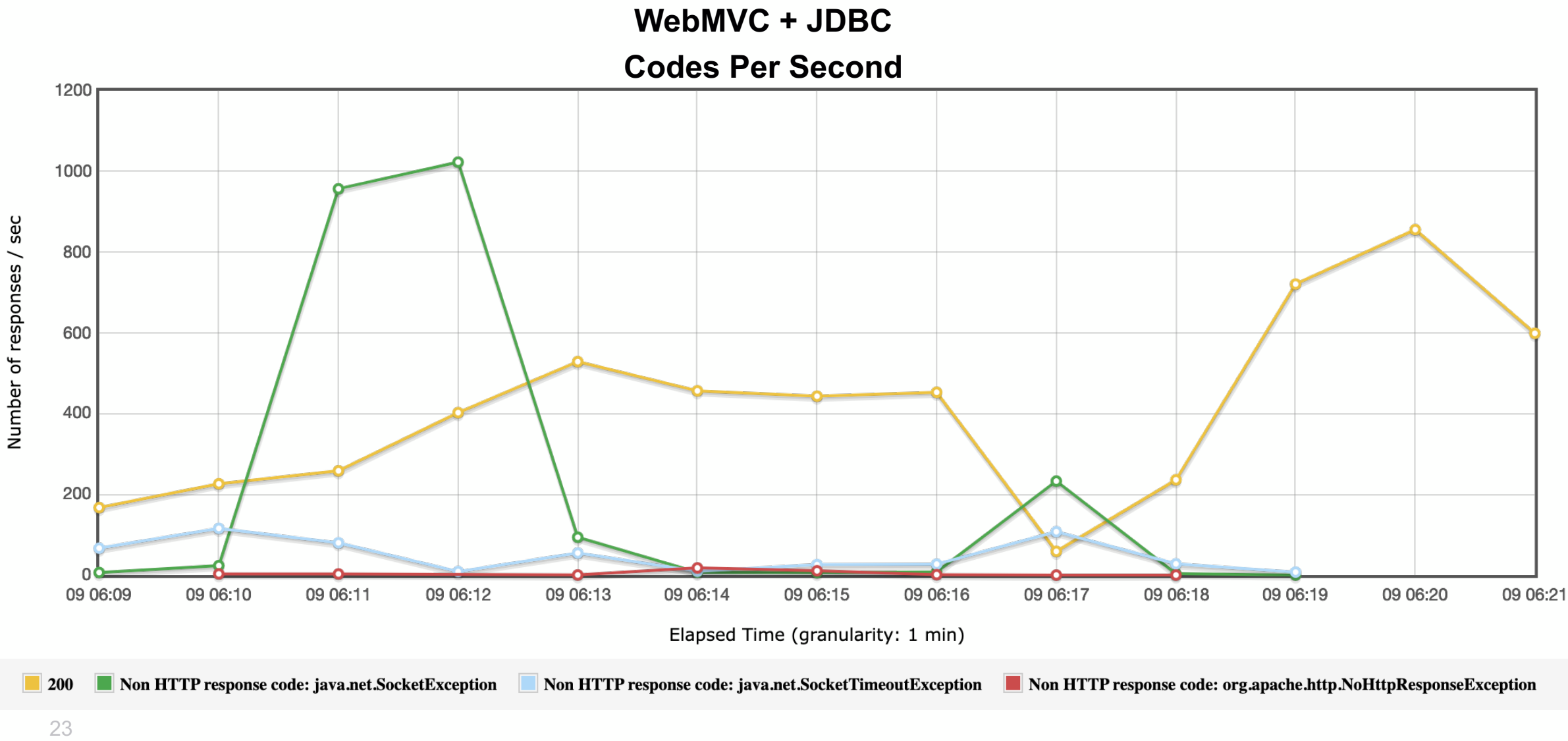


WebFlux + R2DBC



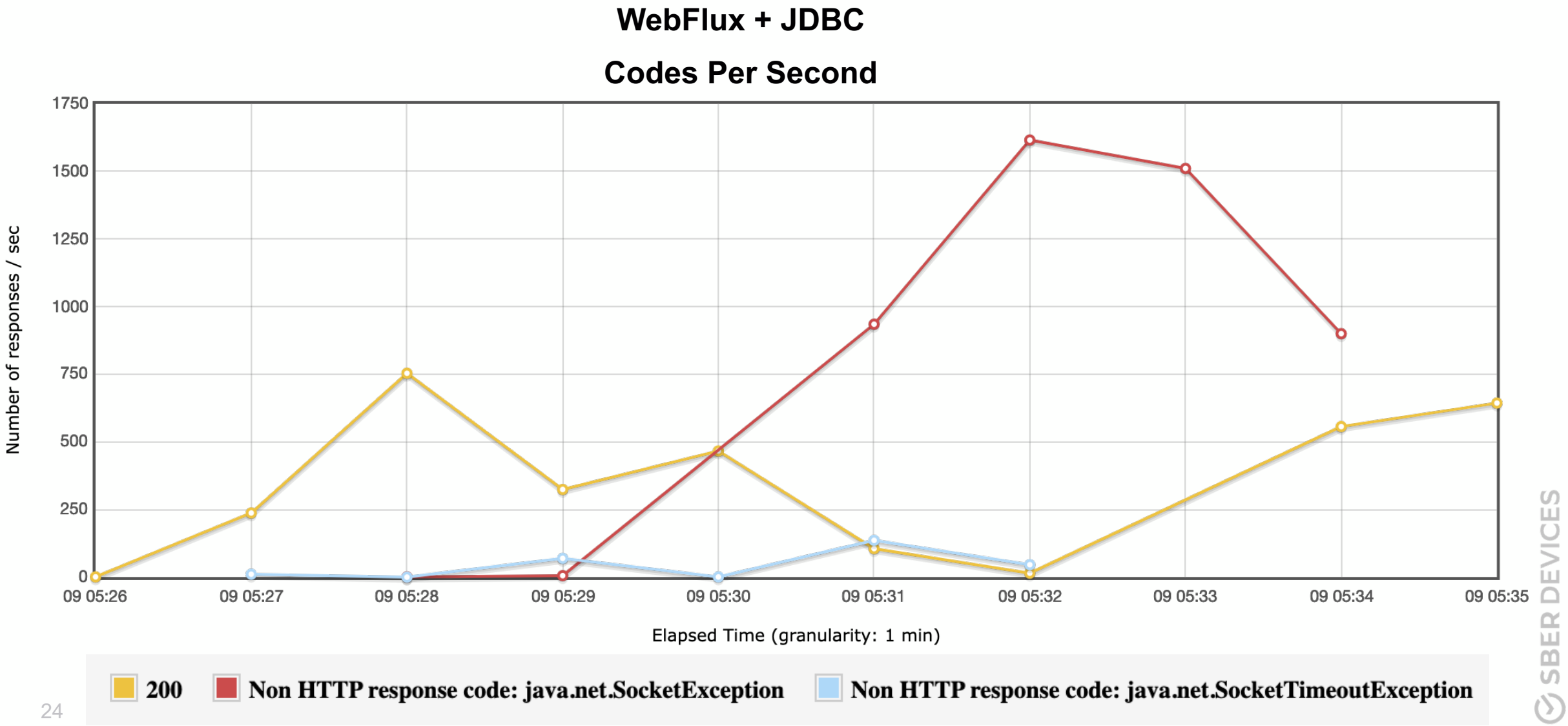
SBER DEVICES

Load test: 5000 users, light requests



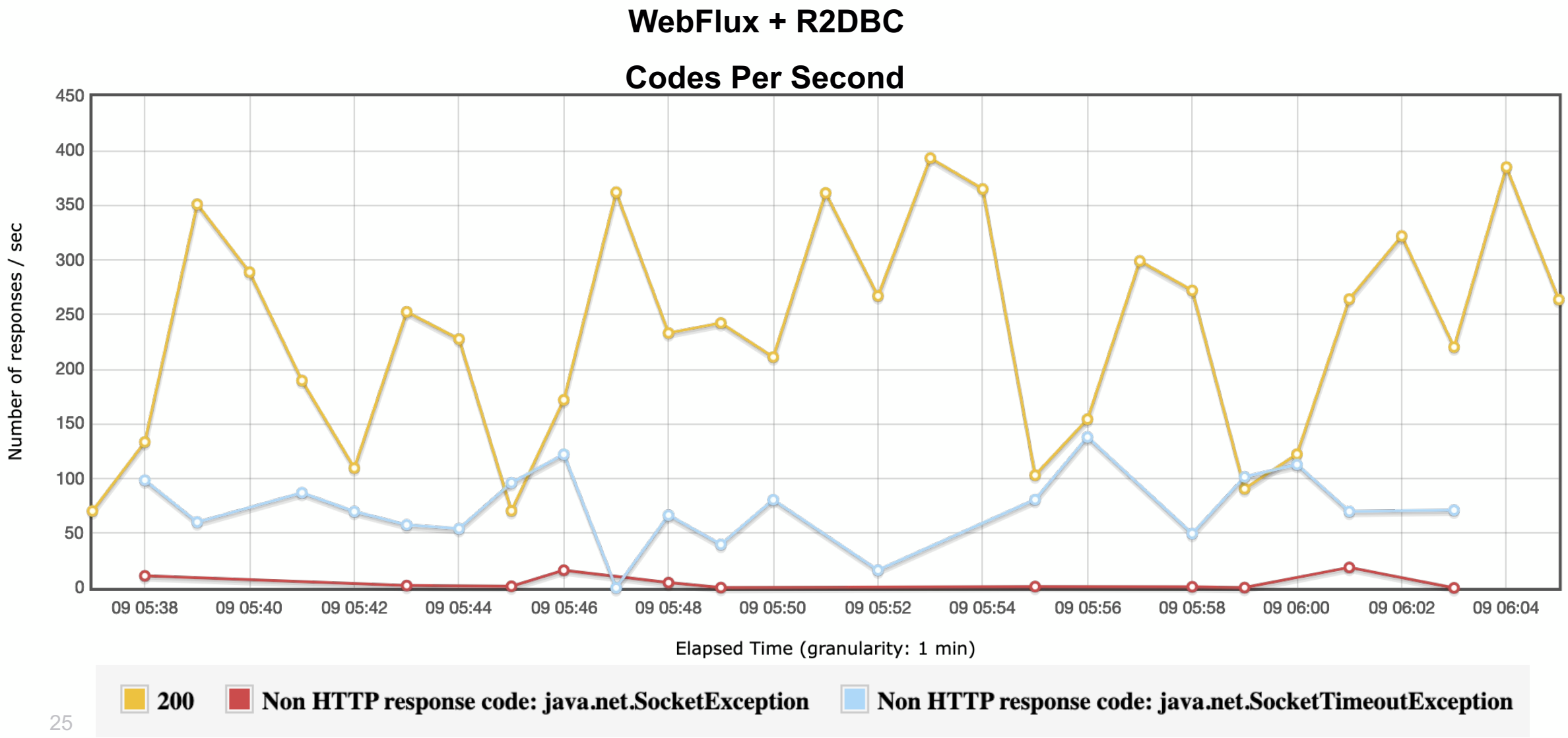
SBER DEVICES

Load test: 5000 users, light requests



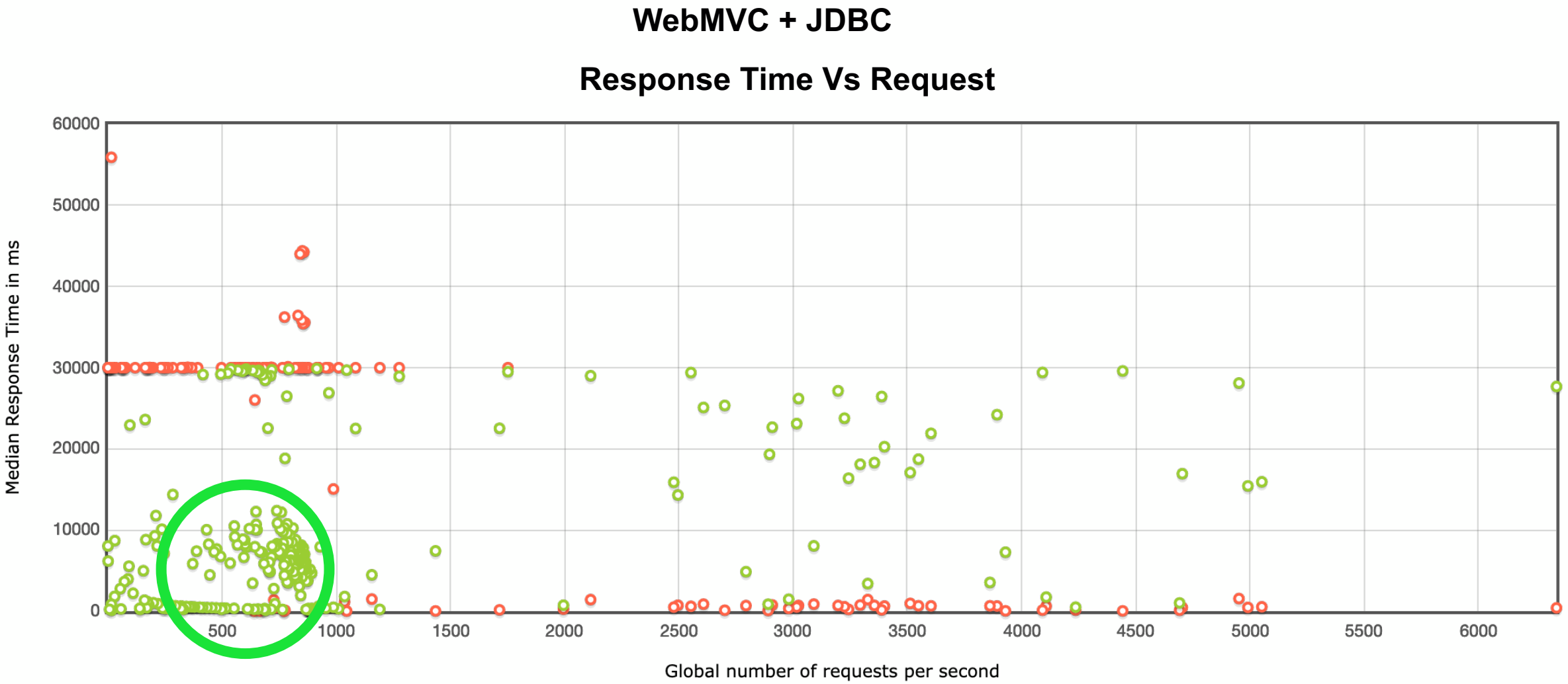
SBER DEVICES

Load test: 5000 users, light requests



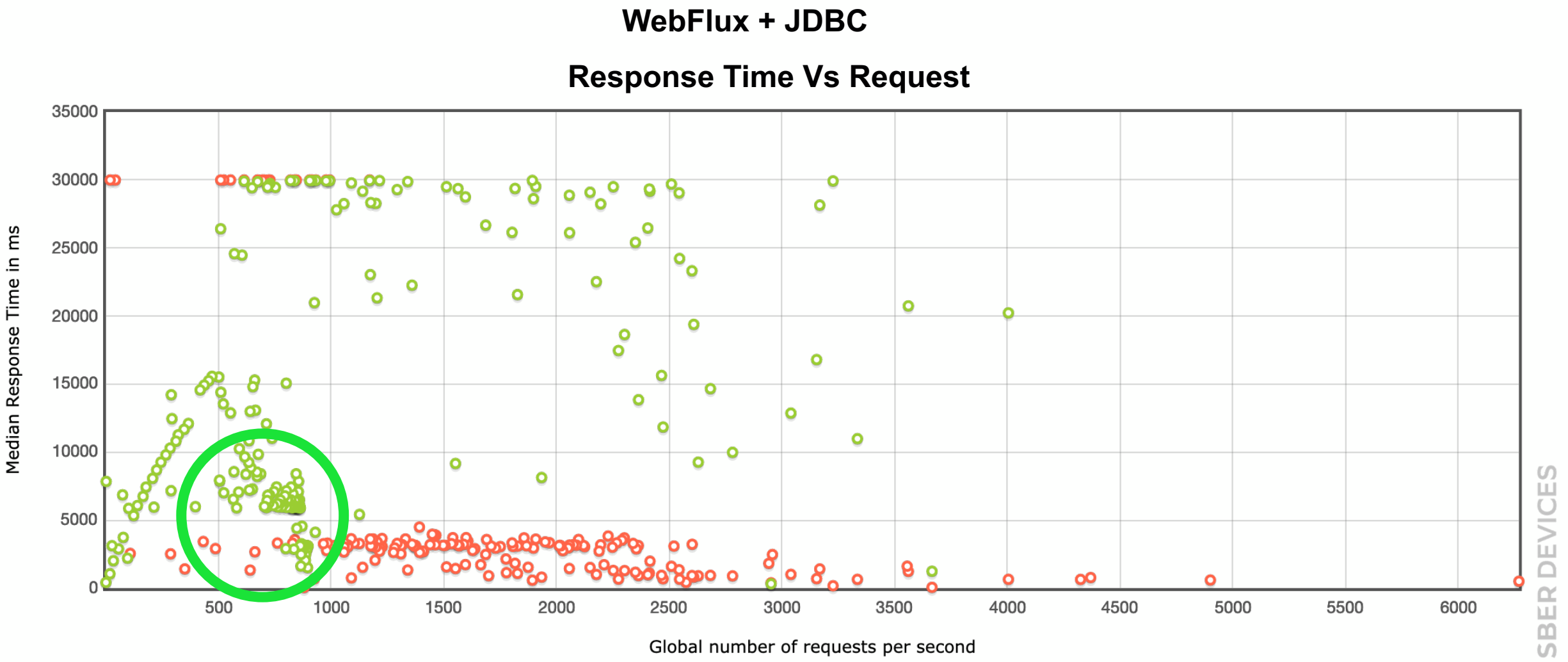
SBER DEVICES

Load test: 5000 users, light requests



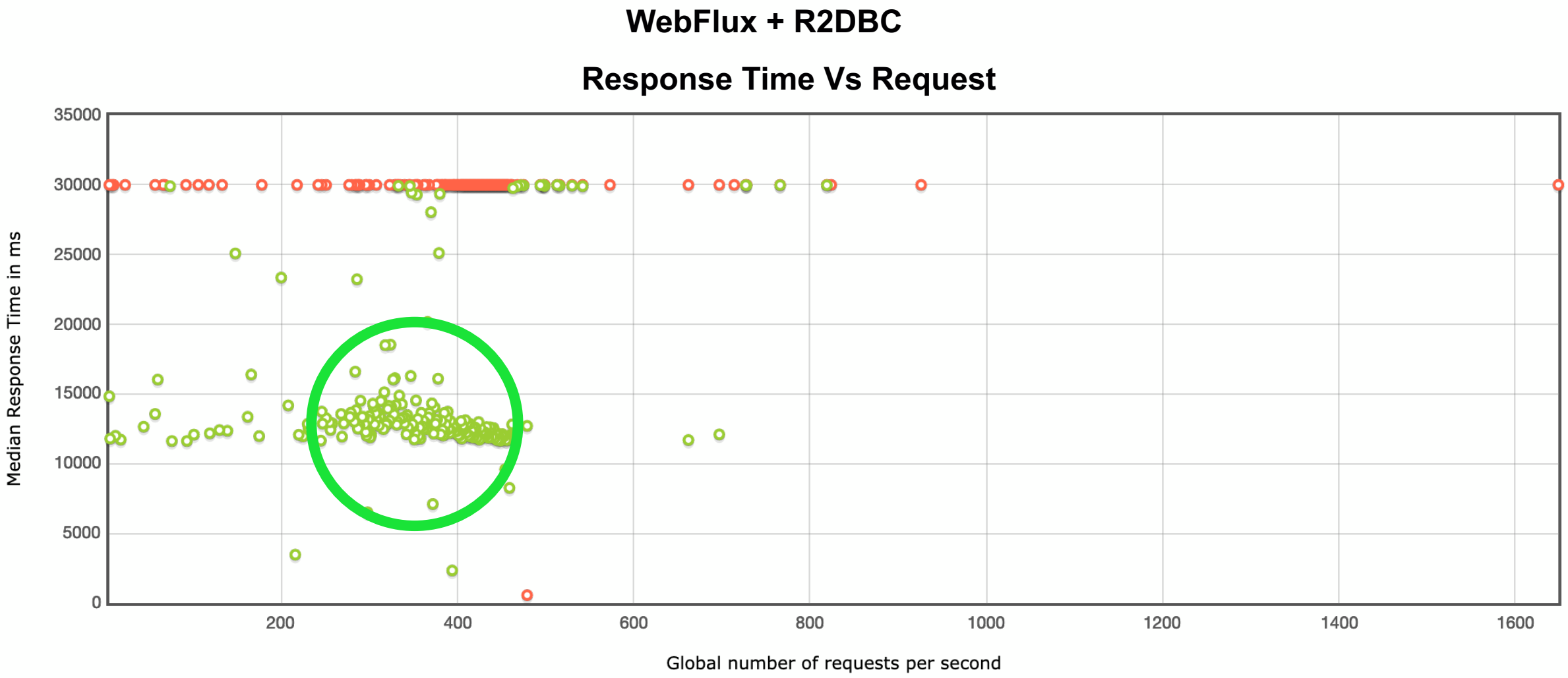
SBER DEVICES

Load test: 5000 users, light requests



SBER DEVICES

Load test: 5000 users, light requests



SBER DEVICES

Load test: summary

- R2DBC works slower than JDBC
- R2DBC responds very well and smoothly to a sharp increase in load
- R2DBC gives more stable response time
- R2DBC handles much higher loads than JDBC
- WebFlux + JDBC doesn't make sense

SBER DEVICES

Spring Data R2DBC

SBER DEVICES

Spring Data R2DBC

```
ConnectionFactory connectionFactory = ...

DatabaseClient client = DatabaseClient.create(connectionFactory);
Flux<String> result =
    client.sql("SELECT firstname FROM PERSON WHERE age > :age")
  .bind("age", 42)
  .map(row -> row.get("firstname", String.class))
  .all();

public interface ConnectionAccessor {
  <T> Mono<T> inConnection(Function<Connection, Mono<T>> action)...
  <T> Flux<T> inConnectionMany(Function<Connection, Flux<T>> action)...
}
```


SBER DEVICES

Spring Data R2DBC: Entity & Repository

```
@Table("person")
public class Person {
  @Id
  @Column("id")
  Long id;

  @Column("age")
  Integer age;

  @Column("firstname")
  String firstName;
}
```

```
@Repository
public interface PersonRepository
    extends R2dbcRepository<Person, Long> {
  Flux<Person> findAllByAgeGreaterThan(Integer age);
}

public interface ReactiveCrudRepository<T, ID>... {
  <S extends T> Mono<S> save(S entity);
  <S extends T> Flux<S> saveAll(Iterable<S> entts);
  Mono<T> findById(ID id);
  ...
}
```

```
Flux<String> result = personRepository.findAllByAgeGreaterThan(42)
  .map(Person::getFirstName)
```


SBER DEVICES

Spring Data R2DBC: transactions

```
public class PersonService {
  PersonRepository personRepository;
  HobbyRepository  hobbyRepository;
  TransactionalOperator txOperator;
  @Transactional
  public Mono<Void> save(Person person, List<Hobby> hobbies) {
    Mono<Person> savedPersonMono = personRepository.save(person);
    Mono<List<Hobby>> savedHobbiesMono =
      savedPersonMono.flatMap(savedPerson -> {
        hobbies.forEach(hobby -> hobby.setPersonId(savedPerson.getId()));
        return hobbyRepository.saveAll(hobbies).collectList();
      });
      return savedHobbiesMono.as(txOperator::transactional).then();
      return savedHobbiesMono.then();
  }
}
```


SBER DEVICES

Reactive transactions: notice

- Requires Reactor Context
- Avoid cancellation of transactional stream: `Flux.next()`, `Flux.take(...)`, cancellation leads to:
  - rollback in Spring 5.3
  - partial inconsistent commit in Spring 5.2

SBER DEVICES

Portability of JDBC operations

SBER DEVICES

Batch operations

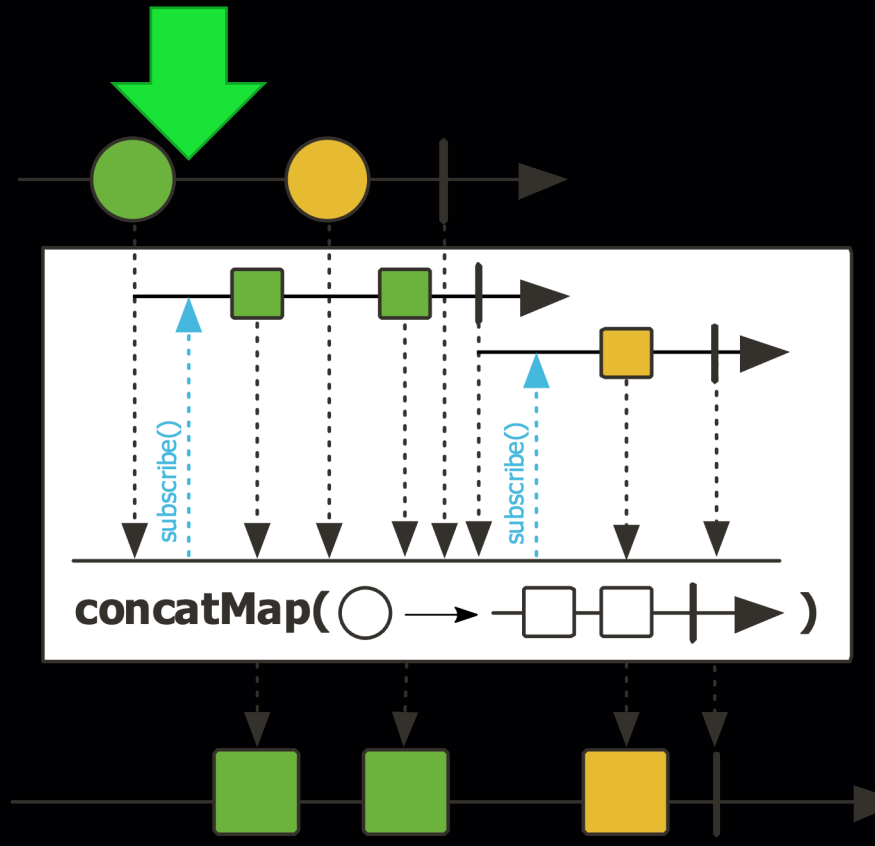
```
@Transactional
public <S extends T> Flux<S> saveAll(Iterable<S> objectsToSave) {
    return Flux.fromIterable(objectsToSave).concatMap(this::save);
}
```

Generation of inners and subscription: this operator waits for one inner to complete before generating the next one and subscribing to it.

Sequential save operation works very slowly.

Provokes misuse

```
@Transactional
public <S extends T> Flux<S> saveAll(Publisher<S>
  objectsToSave) {
    return
  Flux.from(objectsToSave).concatMap(this::save);
}
```



SBER DEVICES

Batch operations: solution attempts

```
//  naive batch
var statement = connection.createStatement(
    "INSERT INTO hobby (person_id, hobby) VALUES ($1, $2)")
  .returnGeneratedValues("id", person_id", ”hobby");
for (String hobby : hobbies) {
    statement.bind("$1", personId).bind("$2", hobby).add();
}
return Flux.from(statement.execute()).flatMap(result -> ... buildHobby(row)));

/* ------------------------------------------------------------ */
//  correct batch
var batch = connection.createBatch();
for (String hobby : hobbies) {
  batch.add(String.format(
    "INSERT INTO hobby (person _id, hobby) VALUES (%s, '%s') RETURNING *",
      personId, hobby));
}
return Flux.from(batch.execute()).flatMap.(result -> ... buildHobby(row)));
```

The same slow sequential save

Does not cache statements

SBER DEVICES

Batch operations: final custom solution

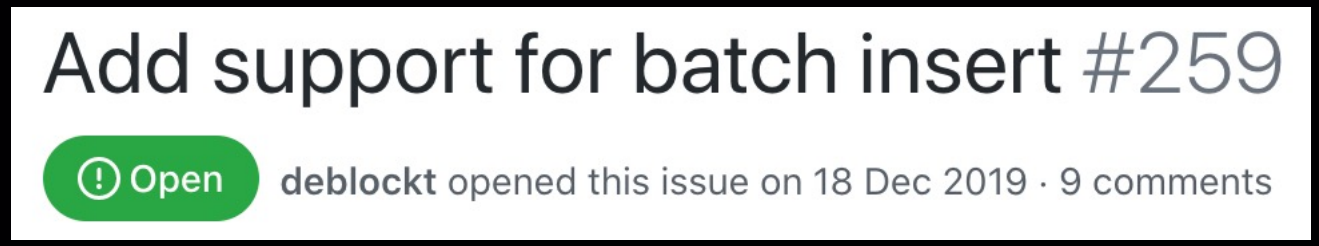
```
public Flux<Hobby> saveAllInBatch(
        Long personId, List<String> hobbies) {

  return databaseClient.sql(
    "insert into hobby(person_id, hobby) " +
    "select :person _id, hobbies.hobby " +
    "from unnest(array[ :hobbies ]) hobbies(hobby) " +
    "returning *")
    .bind("person _id", personId)
    .bind("hobbies", hobbies)
    .as(Hobby.class)
    .fetch()
    .all();
}
```

Pros:

- Works fast
- Caches the statement

Cons:

- Requires manual coding
- Can’t be reused for other (not Postrges) database provider



SBER DEVICES

One-to-one and one-to-many

```
// works not so fast, but works: for small number of elements not so slow
Flux<Person> = personRepository.findAllBy...() // Flux: 10 - 100 elements
  .concatMap(person -> hobbyRepository.findByPersonId(person.getId())
    .doOnNext(person::setHobby).thenReturn(person));


// does not work – deadlock
Flux<Person> = personRepository.findAllBy...() // Flux: 1000 elements
  .concatMap(person -> hobbyRepository.findByPersonId(person.getId())
    .doOnNext(person::setHobby).thenReturn(person));

personRepository.findAllBy...(): // Flux
Person1 -> hobbyRepository.findByPersonId(person.getId())
Person2 -> hobbyRepository.findByPersonId(person.getId())
...
Person128 -> hobbyRepository.findByPersonId(person.getId())
```


SBER DEVICES

One-to-one and one-to-many

- Object mapping is a synchronous process
- Flux result is read from the response stream
- Sub-queries are not possible

Workaround 1: terminate response stream with Flux.collectList()

Workaround 2: use joins or stored procedures

SBER DEVICES

Possible memory leak in PostgreSQL R2DBC driver

```
// possible leak
databaseClient.sql("insert into person(person_id) values(" + personId + ")")...

// good
databaseClient.sql("insert into person(person_id) values(:person_id)")
  .bind("person_id", personId)...
```

```
/**
 * Configure the preparedStatementCacheQueries. The default is {@code -1},
   meaning there's no limit. The value of {@code 0} disables the cache.
  Any other value specifies the cache size.
 *
 * @param preparedStatementCacheQueries the preparedStatementCacheQueries
 * @return this {@link Builder}
 * @throws IllegalArgumentException if {@code username} is {@code null}
 * @since 0.8.1
 */
```


SBER DEVICES

Performance

Spring Data R2DBC two times slower than Spring Data JDBC for Flux response #203

Closed **Aleksandr-Filichkin** opened this issue on 30 Sep 2019

mp911de commented on 15 Oct 2019

Closing this issue as this isn't something we can work on for now.

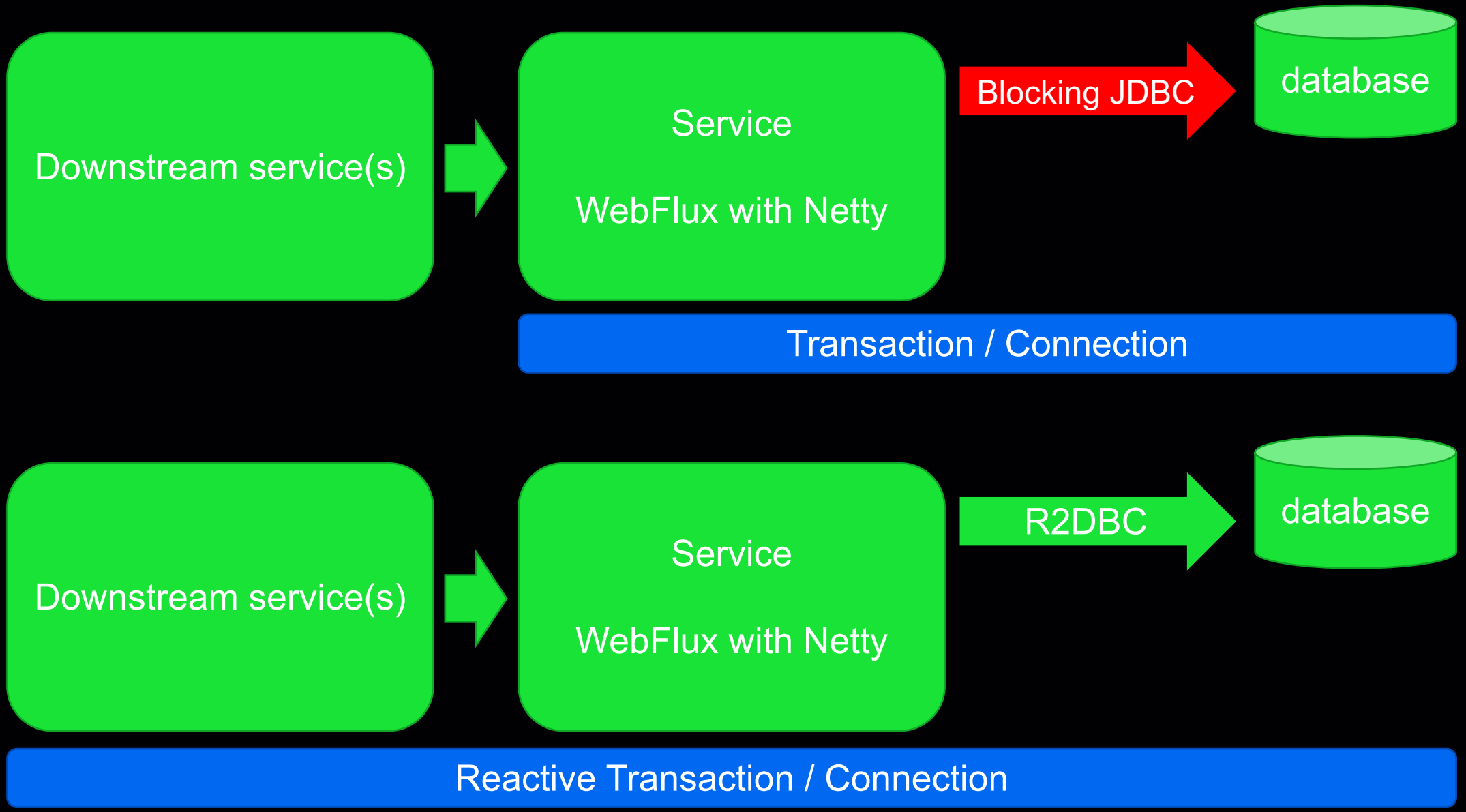
mp911de closed this on 15 Oct 2019

mp911de commented on 15 Oct 2019 Member

You don't buy into reactive programming because you want your queries to run faster. You apply reactive programming patterns to improve your application scalability and resilience.

👍 13 👎 3 😄 1 😕 4 👀 1

More about connections



SBER DEVICES

R2DBC Cookbook

SBER DEVICES

R2DBC Cookbook

1. Close connections!
2. Properly configure connection pool.
3. Avoid extra logic in transaction / connection context.
4. Use variable bindings.
5. Write custom manual queries for batch insert / update operations.
6. Avoid usage of Flux<?> in your API, call Flux.collectList() method on repository level.
7. While using Flux<?> as a result stream from a database avoid transformation of the flux with extra calls to the database, instead use Flux.collectList().
8. Use joins or stored procedures for one-to-one and one-to-many relationships.
9. You can use JDBC along with R2DBC.

SBER DEVICES

JDBC with R2DBC: configuration

```
// along with spring-boot-starter-data-r2dbc for R2dbc*AutoConfiguration
@Configuration
public class JdbcConfig {
  @Bean
  @ConfigurationProperties(prefix = "spring.datasource.hikari")
  public HikariDataSource springDataSource() {
    return new HikariDataSource();
  }

  @Bean
  public PlatformTransactionManager transactionManager(
      HikariDataSource dataSource) {
    return new JdbcTransactionManager(dataSource);
  }
}
```


SBER DEVICES

JDBC with R2DBC: usage

```
public class R2dbcService {
  @Transactional(transactionManager = "connectionFactoryTransactionManager")
  public Mono<Object> save(...) { ... }
}

public class JdbcService {
  @Transactional(transactionManager = "transactionManager")
  public Object save(...) { ... }
}

public class ReactiveJdbcService {
  public Mono<Object> save(...) {
    return Mono.fromCallabe(() -> jdbcService.save(...))
      .subscribeOn(scheduler);
    // return Mono.just(() -> jdbcService.save(...)); // very bad
  }
}
```


SBER DEVICES

Takeaways

- Build your expectations correctly.
  - A reactive approach doesn’t speed things up.
  - It allows you to build scalable solutions.
- Not all JDBC practices are relevant for R2DBC.
  - If your team is used to JDBC, then it will have to relearn.
  - Avoid typical mistakes.
- You can't be half reactive.
  - If you use Spring WebFlux for scalability, you are forced to use R2DBC.
  - Otherwise, you don’t need WebFlux.

SBER DEVICES

Q & A

SBER DEVICES

SBER DEVICES