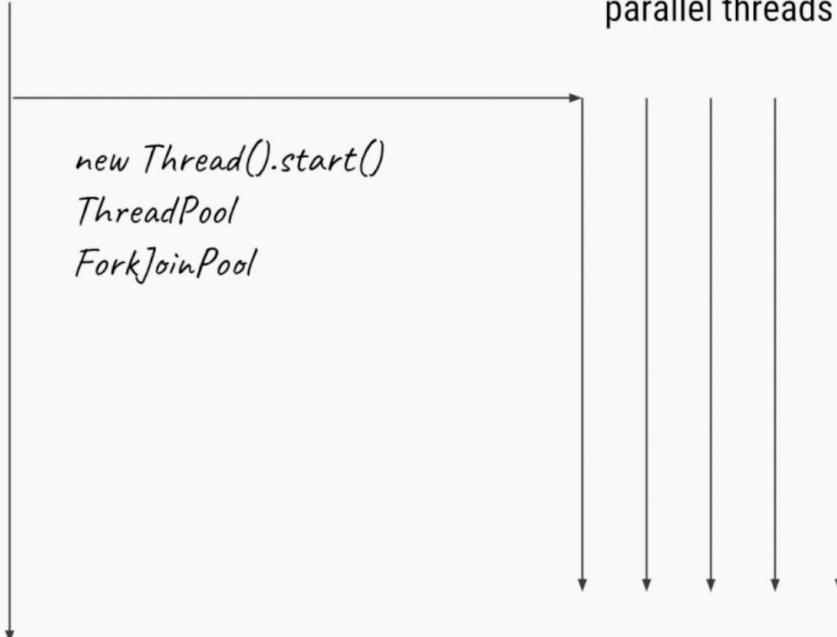


Java - Async Programming

Need for speed

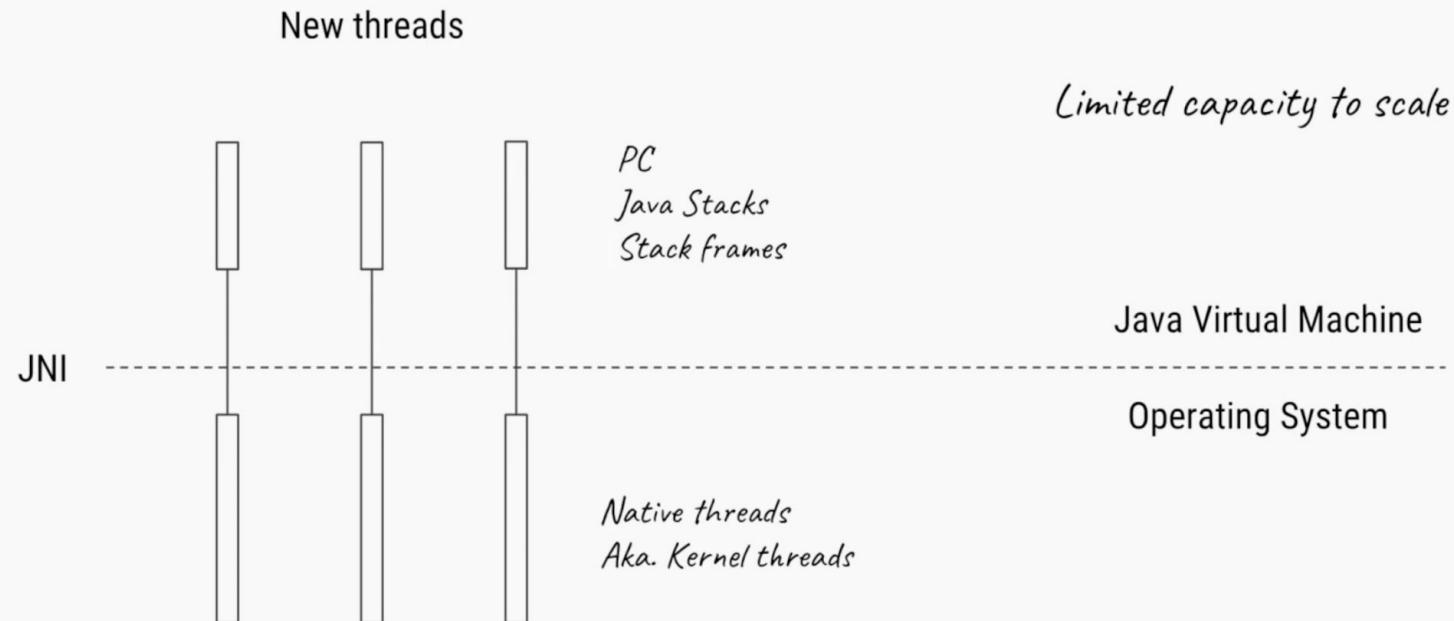
main thread

parallel threads



CPU

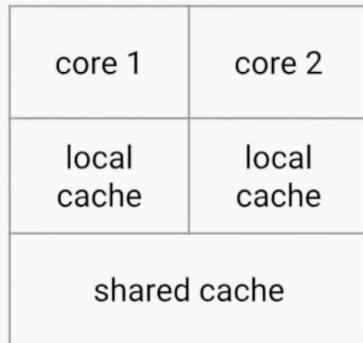
Java thread = OS thread



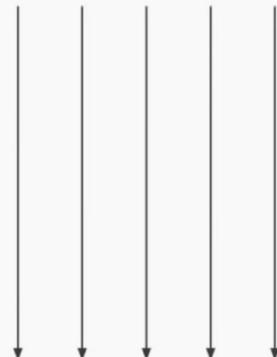
Other issues too...

Scheduling Overhead,

Data Locality



CPU

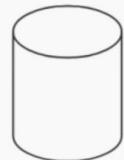


IO operations require more threads

main thread

Limited capacity to scale IO ops

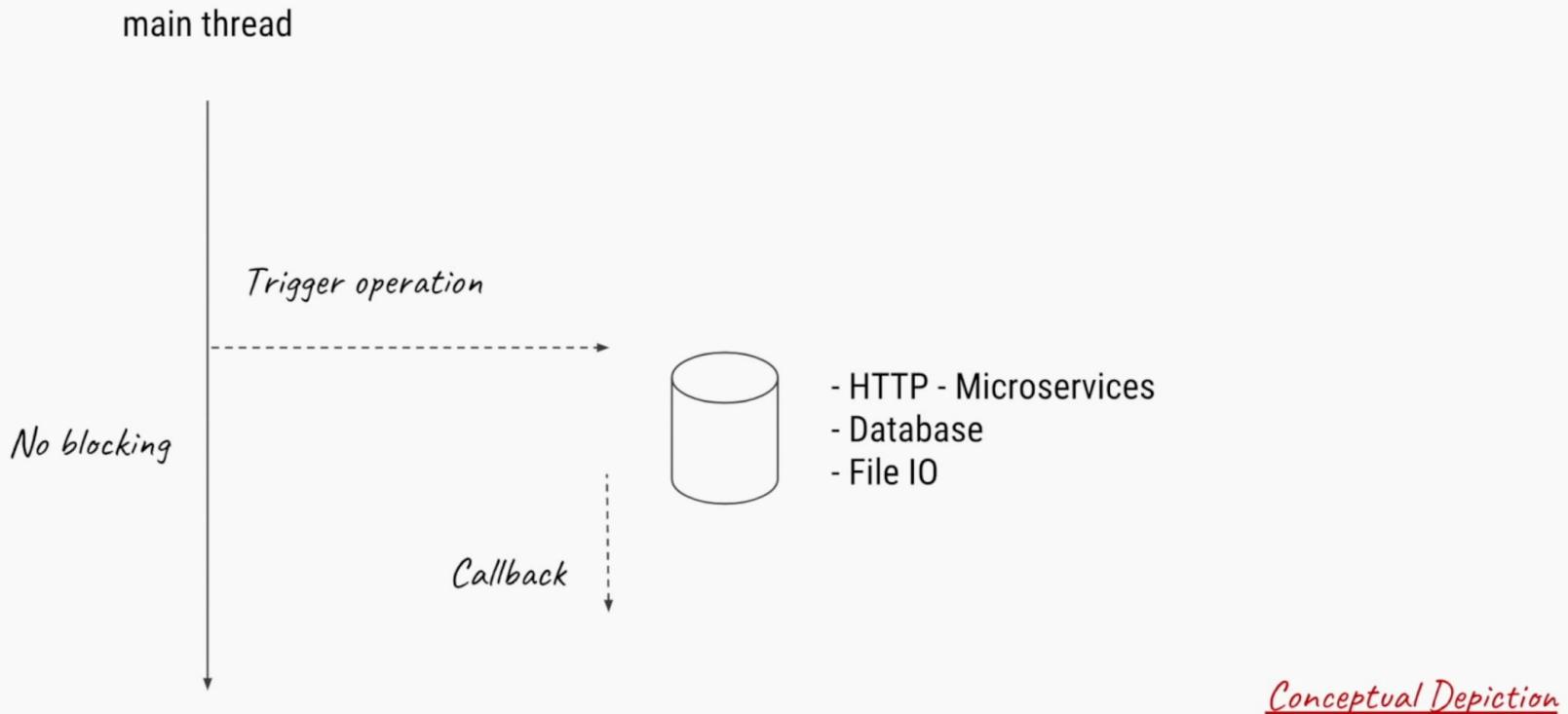
*Blocked
(wait state)*



- Network (HTTP / DB)
- File IO

Runnable

Non-blocking IO



Synchronous API

```
for (Integer id : employeeIds) {  
  
    // Step 1: Fetch Employee details from DB  
    Future<Employee> future = service.submit(new EmployeeFetcher(id));  
    Employee emp = future.get(); // blocking  
  
    // Step 2: Fetch Employee tax rate from REST service  
    Future<TaxRate> rateFuture = service.submit(new TaxRateFetcher(emp));  
    TaxRate taxRate = rateFuture.get(); // blocking  
  
    // Step 3: Calculate current year tax  
    BigDecimal tax = calculateTax(emp, taxRate);  
  
    // Step 4: Send email to employee using REST service  
    service.submit(new SendEmail(emp, tax));  
}
```

Problem

Expensive Threads

&

Blocking IO Ops

=

Limited Scalability

Solution

Non-blocking IO
&
Asynchronous API

Asynchronous API - Callbacks

```
for (Integer id : employeeIds) {  
    CompletableFuture.supplyAsync(() -> fetchEmployee(id))  
        .thenApplyAsync(employee -> fetchTaxRate(employee))  
        .thenApplyAsync(taxRate -> calculateTax(taxRate))  
        .thenAcceptAsync(taxValue -> sendEmail(taxValue));  
}
```

Callback chaining (similar to JS)

Java NIO

- Buffers, Channels, Selectors
- Low-level API for asynchronous / non-blocking IO
- Applicable for Files, Sockets,
- Listener based (callbacks)

```
ByteBuffer buffer = ByteBuffer.allocate(1024);

Path path = Paths.get("/home/file2");
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.READ);

fileChannel.read(buffer, position: 0, buffer, new CompletionHandler<Integer, ByteBuffer>() {

    @Override
    public void completed(Integer result, ByteBuffer data) {           <= Callback
        // process data
    }
})
```

Servlets 3.0 - Allows more concurrent requests

```
@WebServlet(urlPatterns={"/user"}, asyncSupported=true)
public class UserAsyncServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) {

        final AsyncContext context = request.startAsync();
        context.start(new Runnable() {
            public void run() {
                // make the network call
                ServletResponse response = context.getResponse();
                // print to the response
                context.complete();
            }
        });
    }
}
```

Run operations in
separate thread

Immediately return
to serve other requests

Servlets 3.1 - Efficient thread utilization

```
@WebServlet(urlPatterns={"/user"}, asyncSupported=true)
public class UserAsyncServlet_2 extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) throws IOException {

        final AsyncContext context = request.startAsync();
        ServletInputStream input = request.getInputStream();
        input.setReadListener(new ReadListener() {

            @Override
            public void onAllDataRead() throws IOException {
                // data available, process
                // write to output
            }
        });
    }
}
```

Input and Output are
non-blocking

Spring 5.0 Webflux

Spring - Reactive Programming

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Mono<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

When to use?

Careful considerations

Advantages

- Efficient CPU utilization
- Scalability / High-throughput
- Data locality & less context switches
- Reactive
 - Live/Hot source of values
 - Backpressure

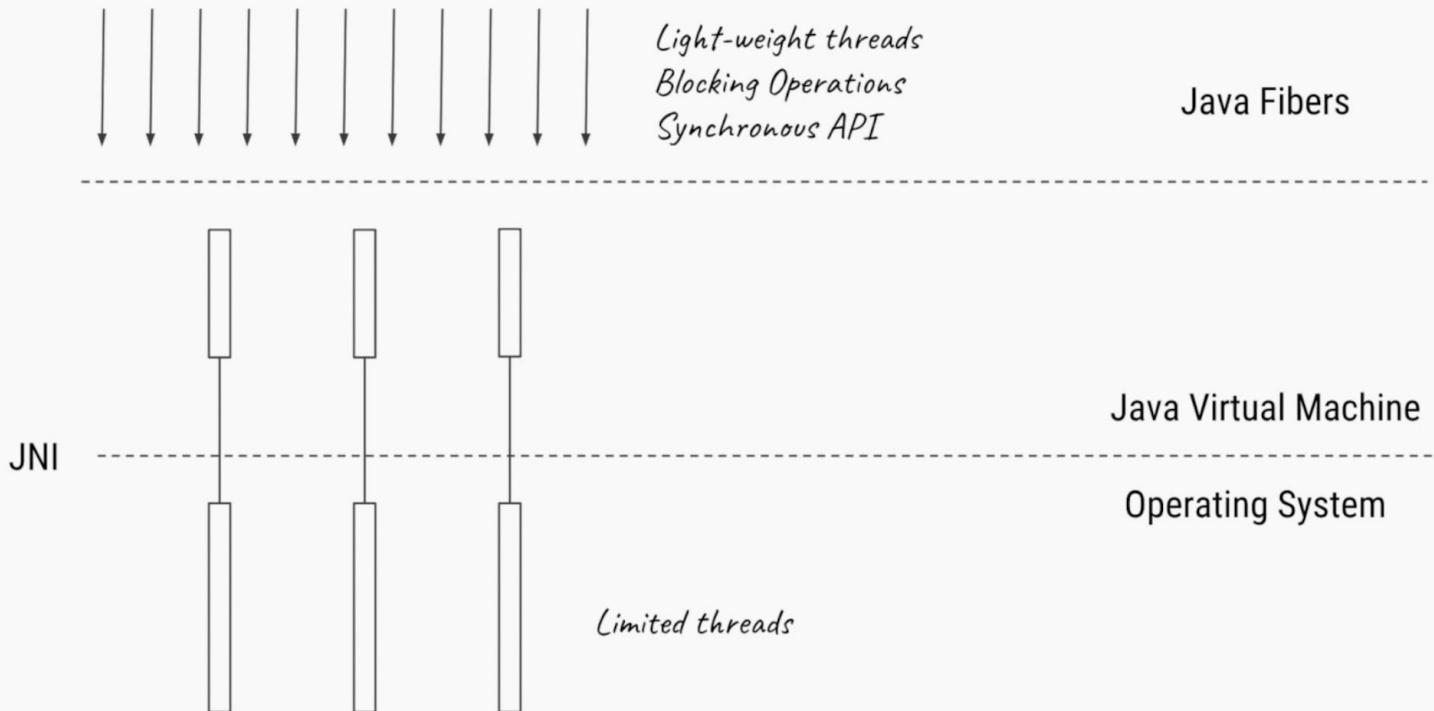
Issues

- Hard to write and reason about
- CPU bound flows
- Hard to debug / Stack trace
- Hard to write tests
- End-to-end async / non-blocking required

What we really
want?

Lightweight
Threads
&
Synchronous
Programming

Java Fibers





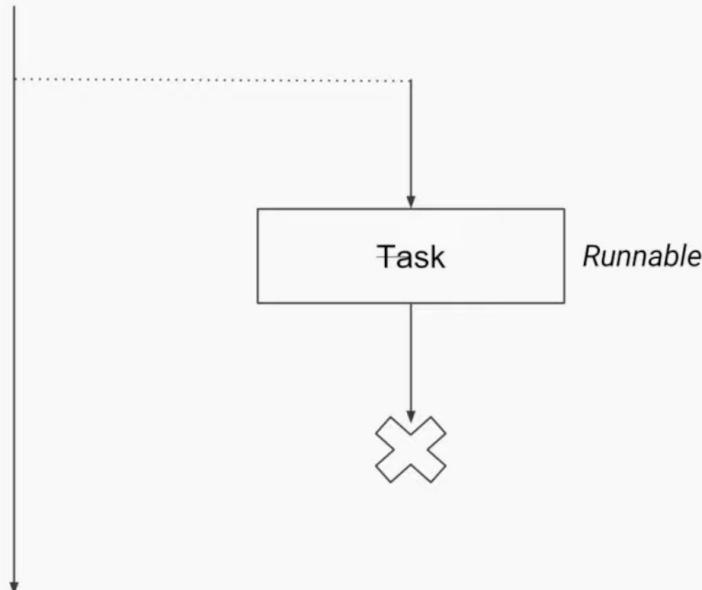
CompletableFuture

What is it used for?

Perform possible asynchronous (non-blocking) computation and trigger dependant computations which could also be asynchronous.

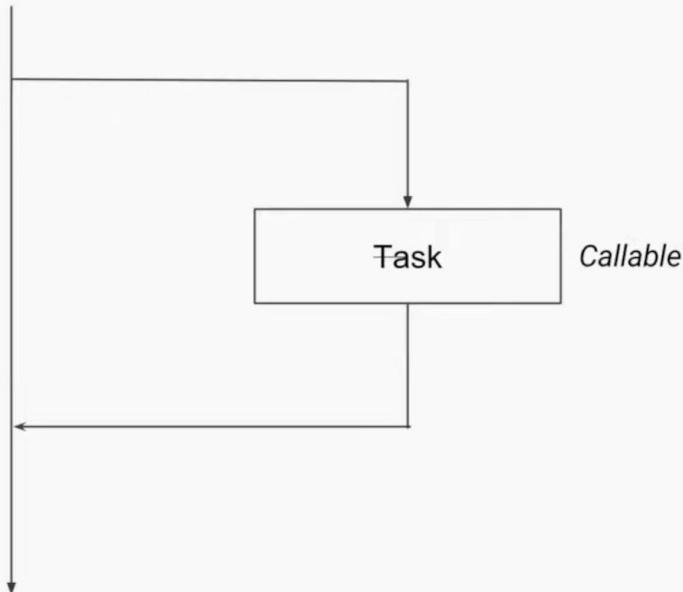
main thread

separate thread



main thread

separate thread



Working with Callable / Futures

```
public static void main(String[] args) {  
  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );  
  
    // submit task and accept the placeholder object for return value  
    Future<Integer> future = service.submit(new Task());  
  
    try {  
        // get the task return value  
        Integer result = future.get(); // blocking  
        System.out.println("Result from the task is " + result);  
  
    } catch (InterruptedException | ExecutionException e) {  
        e.printStackTrace();  
    }  
}  
  
static class Task implements Callable<Integer> {  
    @Override  
    public Integer call() {  
        return new Random().nextInt();  
    }  
}
```

main thread

```
// task implements Callable  
Future f = service.submit(new Task());
```



placeholder

Once result is ready



Set value of placeholder

thread-pool

blocking queue

task1	task2	task3	task4
-------	-------	-------	-------	-------

Fetch next task
from queue

Execute it

t0

t1

t2

t9

main thread

```
// task implements Callable  
Future f = service.submit(new Task());
```

blocked

f.get();

3

Set value of placeholder

runnable

Run dependant task

thread-pool

blocking queue

task1	task2	task3	task4
-------	-------	-------	-------	-------

Fetch next task
from queue

Execute it

t0

t1

t2

t9

main thread

```
for 1..4  
    service.submit(new Task());
```

blocked

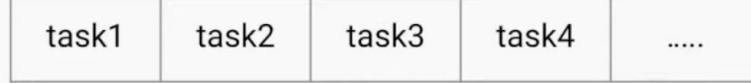


Runnable

Run dependant task

thread-pool

blocking queue



Fetch next task
from queue

Execute it

t0

t1

t2

t9

There is a problem!

main thread

```
for 1..4  
    service.submit(new Task());
```

↓
blocked



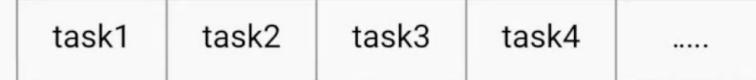
↑
Runnable



Run dependant task

thread-pool

blocking queue



Fetch next task
from queue

Execute it

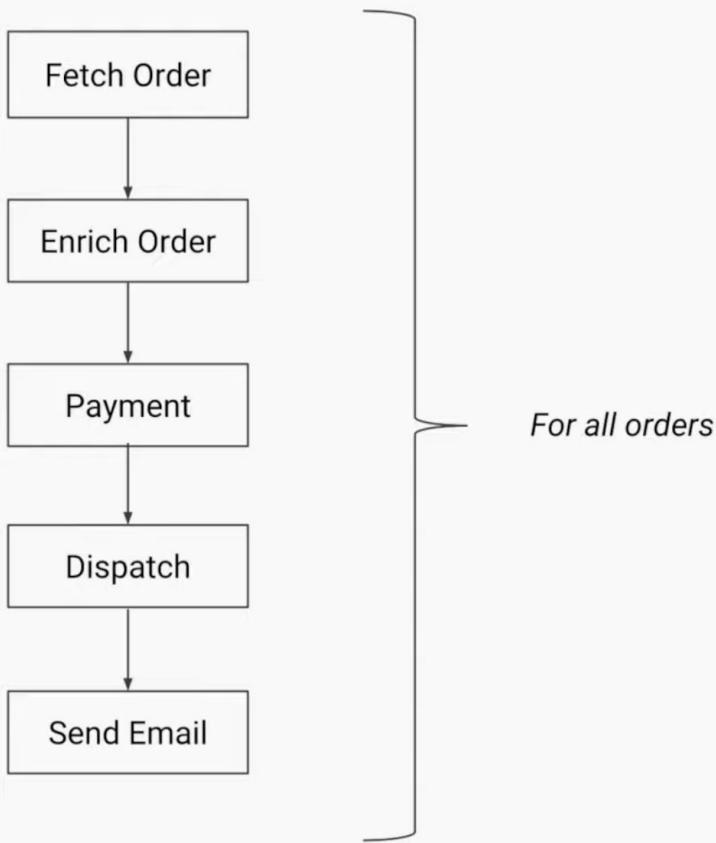
t0

t1

t2

t9

There is a problem!



```
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );

try {

    Future<Order> future = service.submit(getOrderTask());
    Order order = future.get(); // blocking

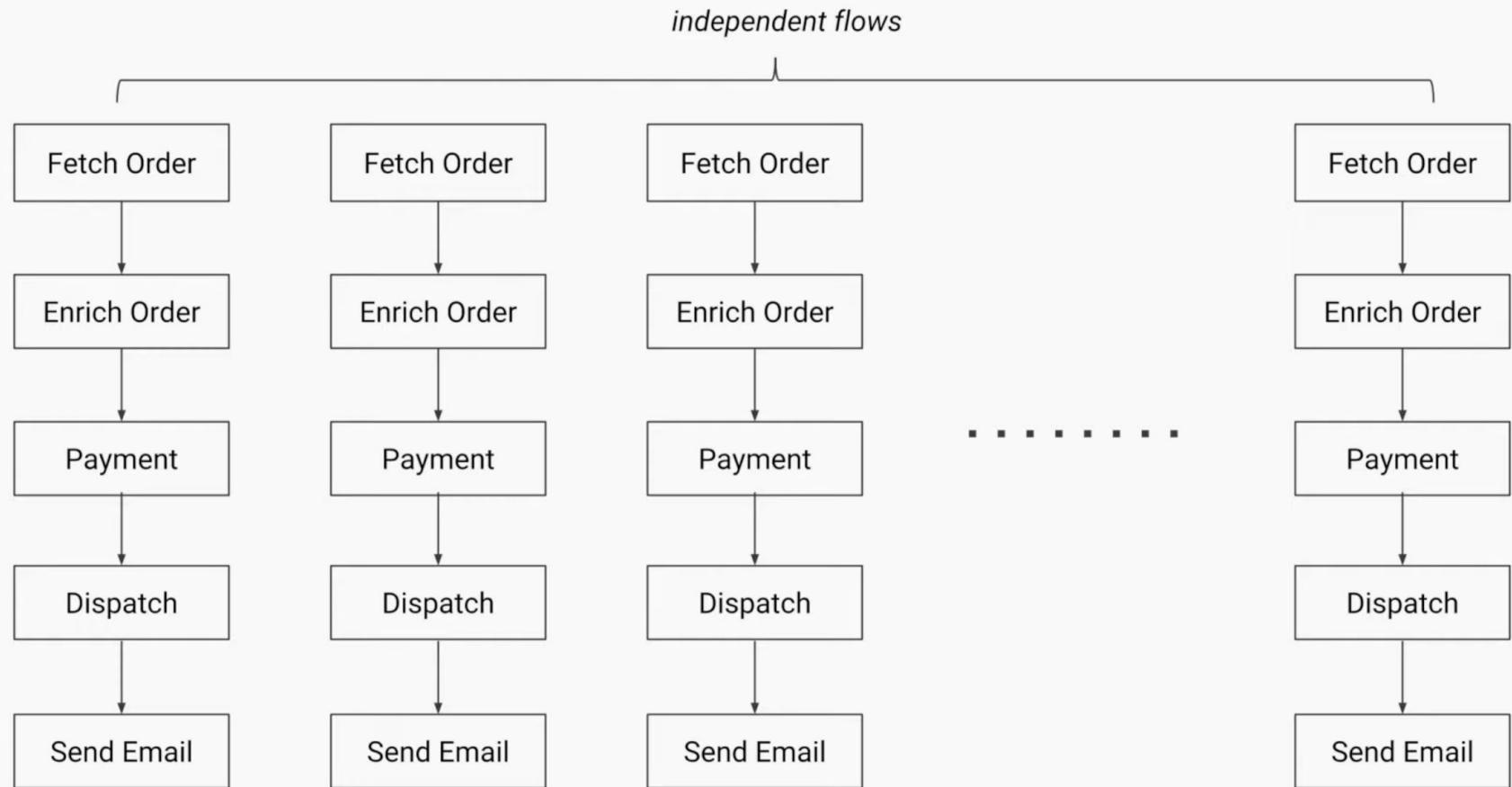
    Future<Order> future1 = service.submit(enrichTask(order));
    order = future1.get(); // blocking

    Future<Order> future2 = service.submit(performPaymentTask(order));
    order = future2.get(); // blocking

    Future<Order> future3 = service.submit(dispatchTask(order));
    order = future3.get(); // blocking

    Future<Order> future4 = service.submit(sendEmailTask(order));
    order = future4.get(); // blocking

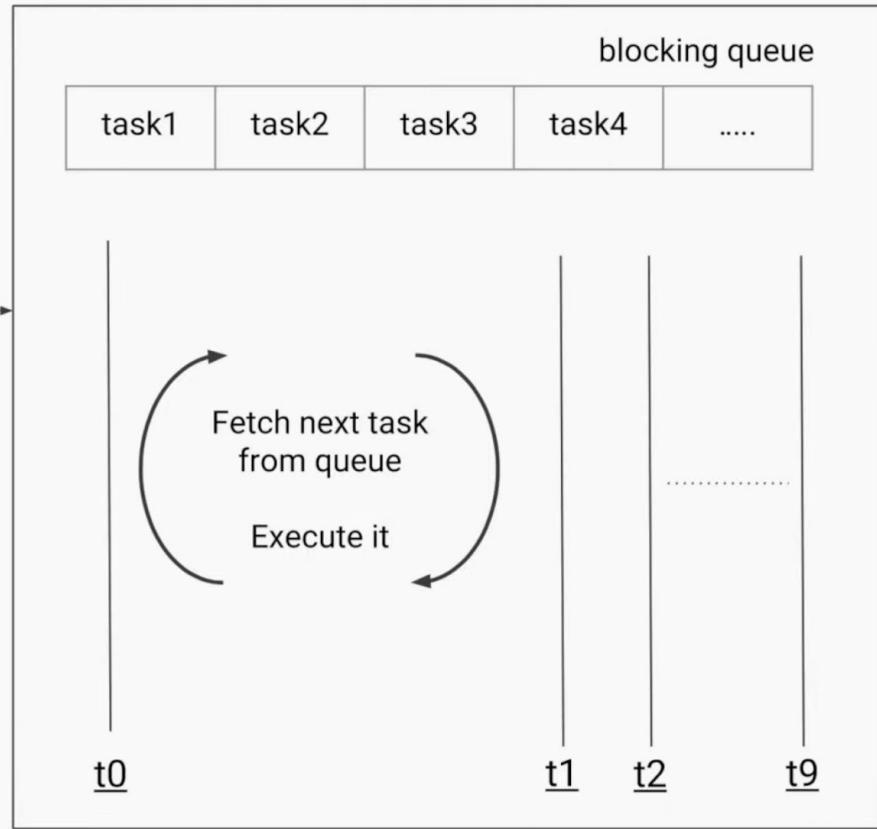
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```



main thread

for 1..n
Run the task
Once done, run the dependant task
Once done, runs its dependant task
And so on.

thread-pool



main thread

for 1..n
Run the task
Once done, run the dependant task
Once done, runs its dependant task
And so on.

thread-pool

I don't care how

Don't bother me

```
for (int i = 0; i < 100; i++) {  
    CompletableFuture.supplyAsync(() -> getOrder())  
        .thenApply(order -> enrich(order))  
        .thenApply(order -> performPayment(order))  
        .thenApply(order -> dispatch(order))  
        .thenAccept(order -> sendEmail(order));  
}  
}
```

Spring WebFlux

Simple web request flow

```
@RestController
public class MyRestController {

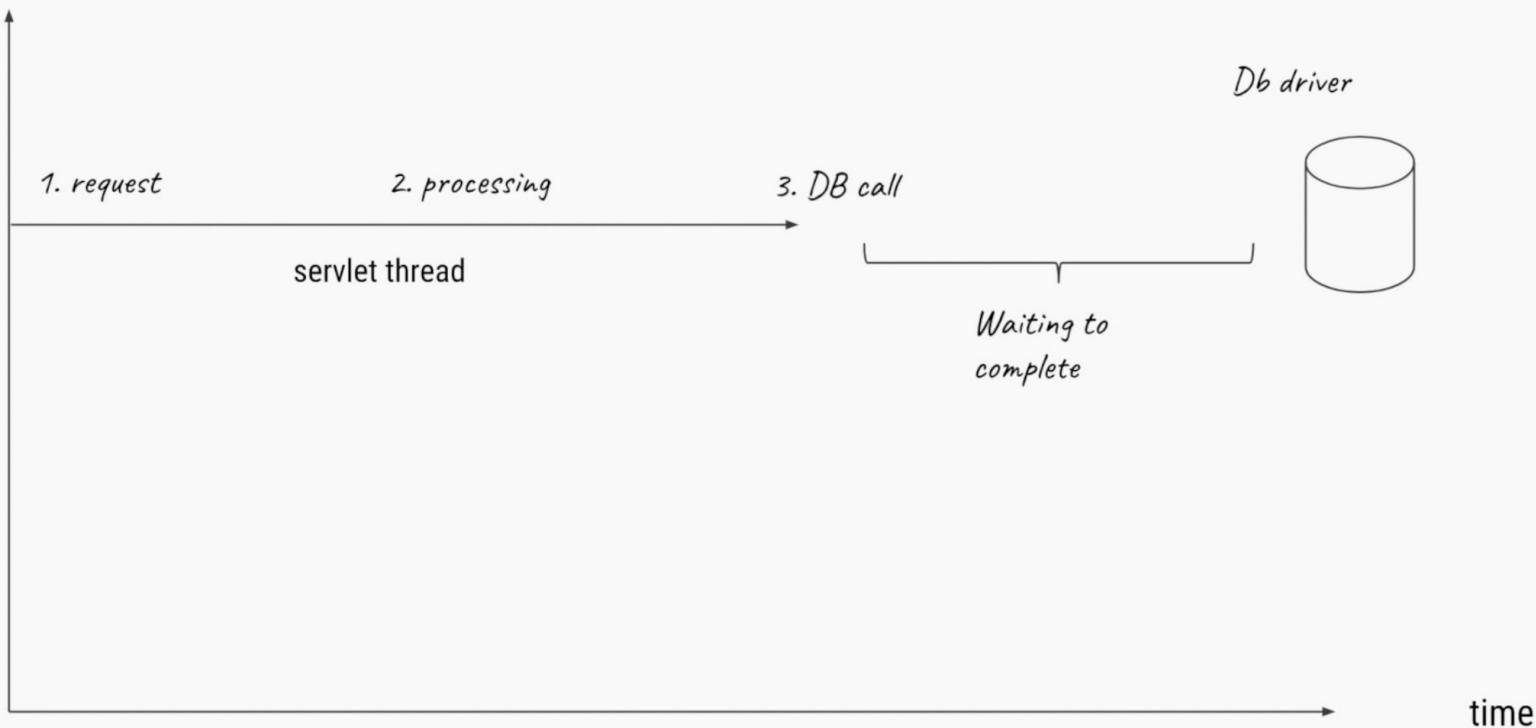
    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public User getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

Thread Perspective

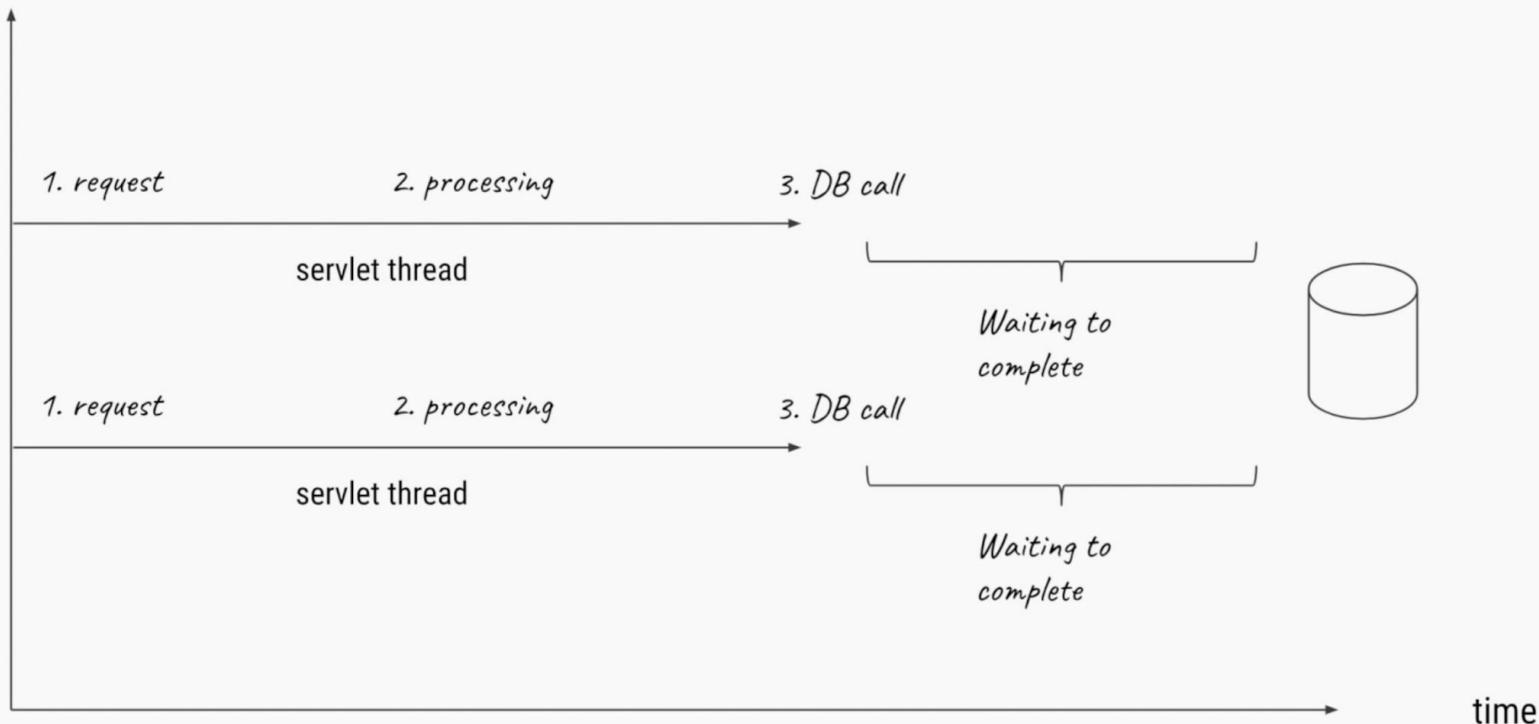
Web requests with IO

threads

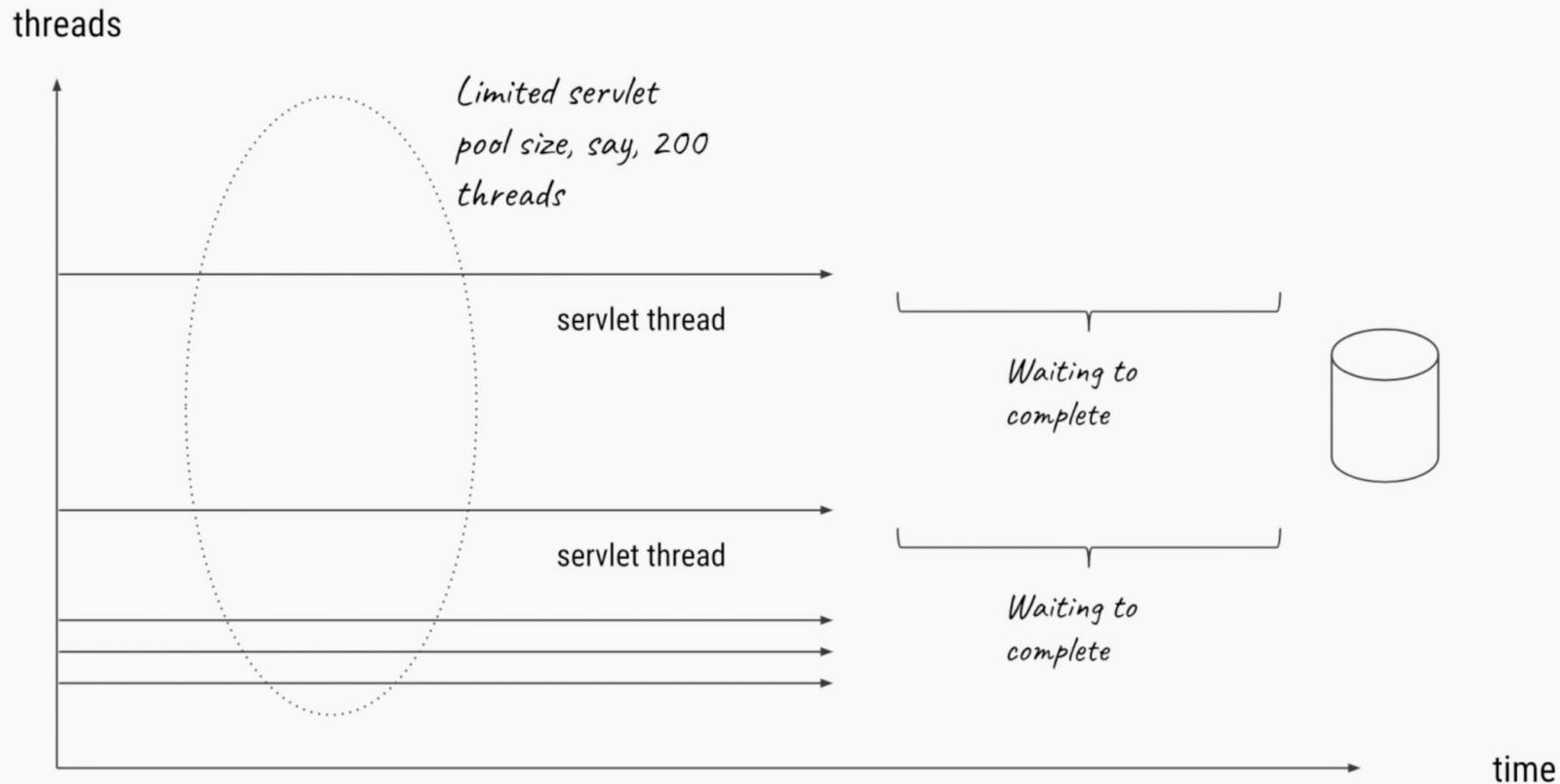


Web requests with IO

threads

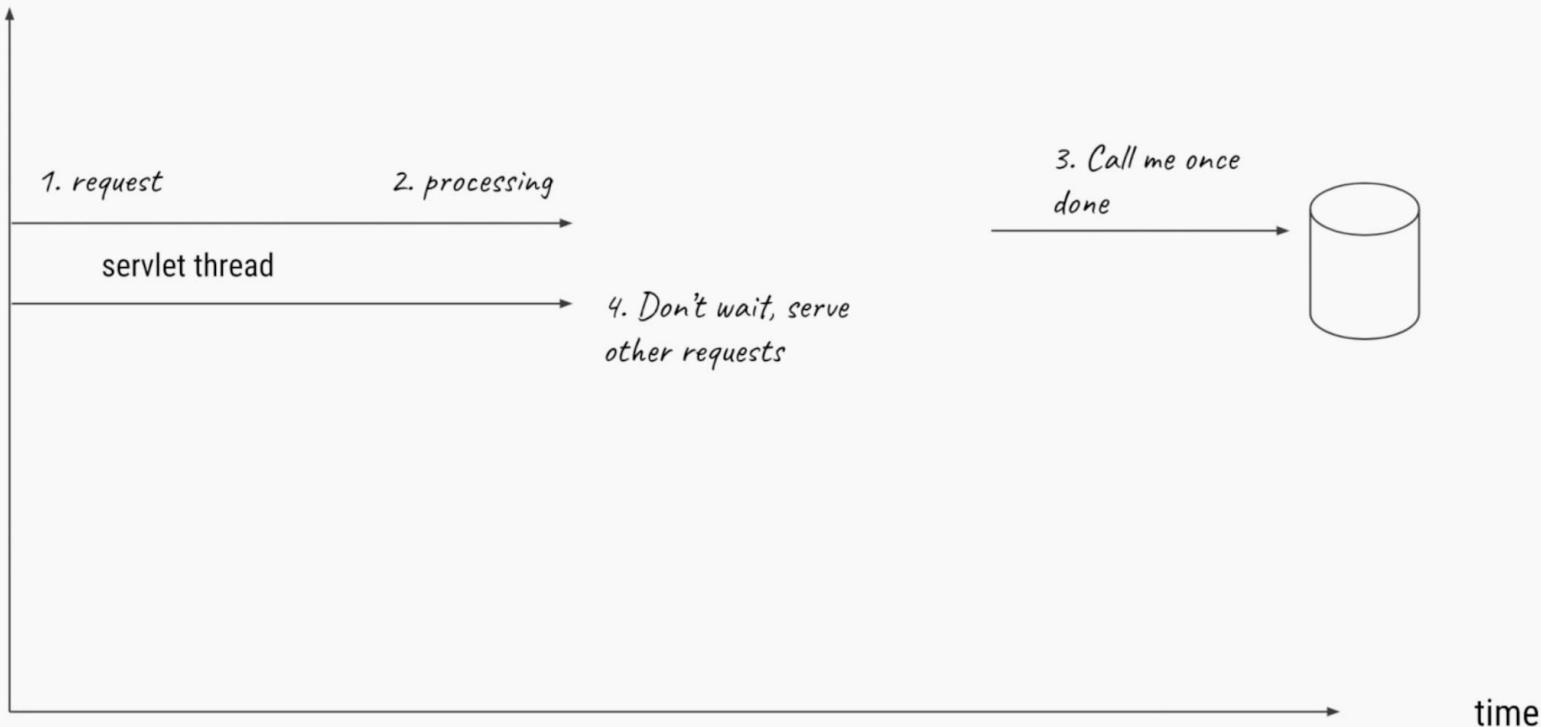


Web requests with IO



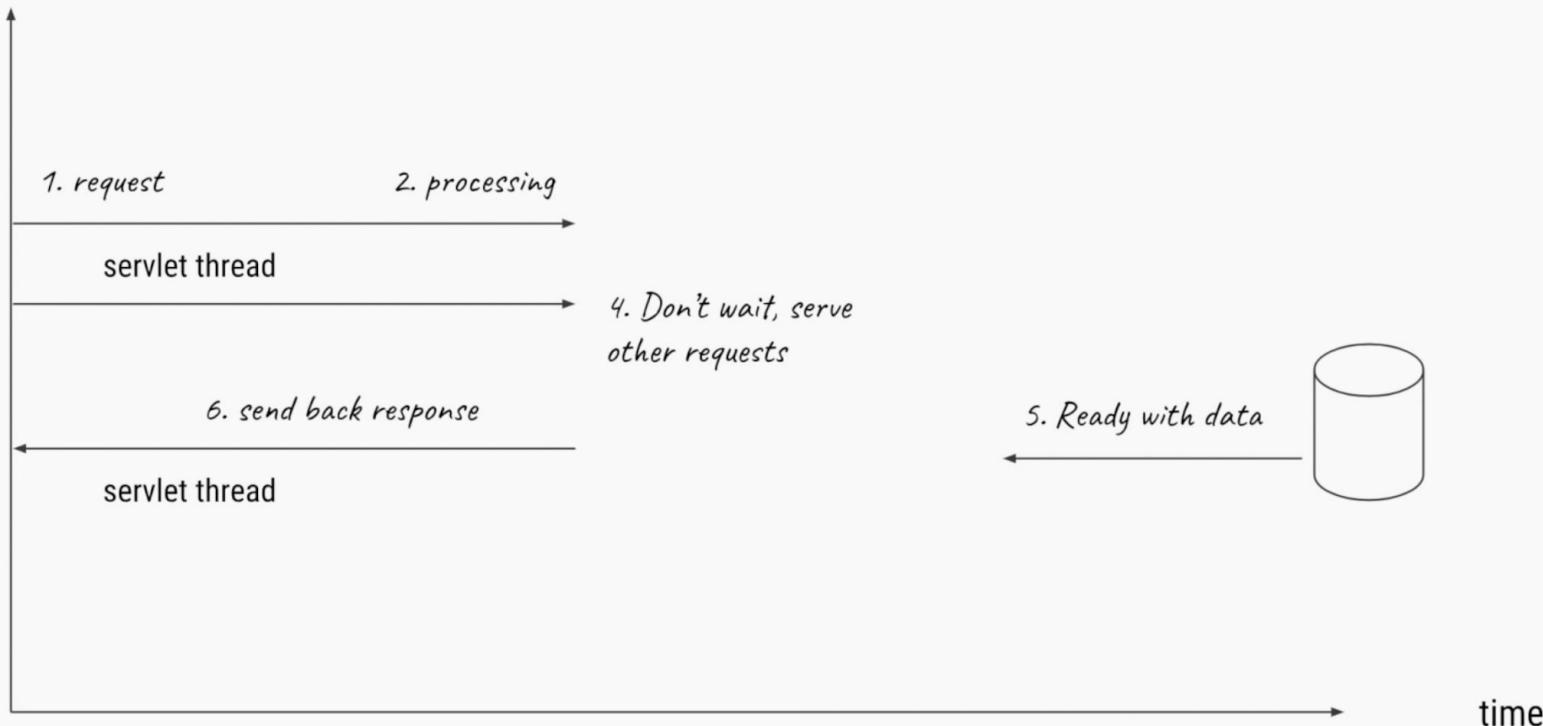
Web requests with IO

threads



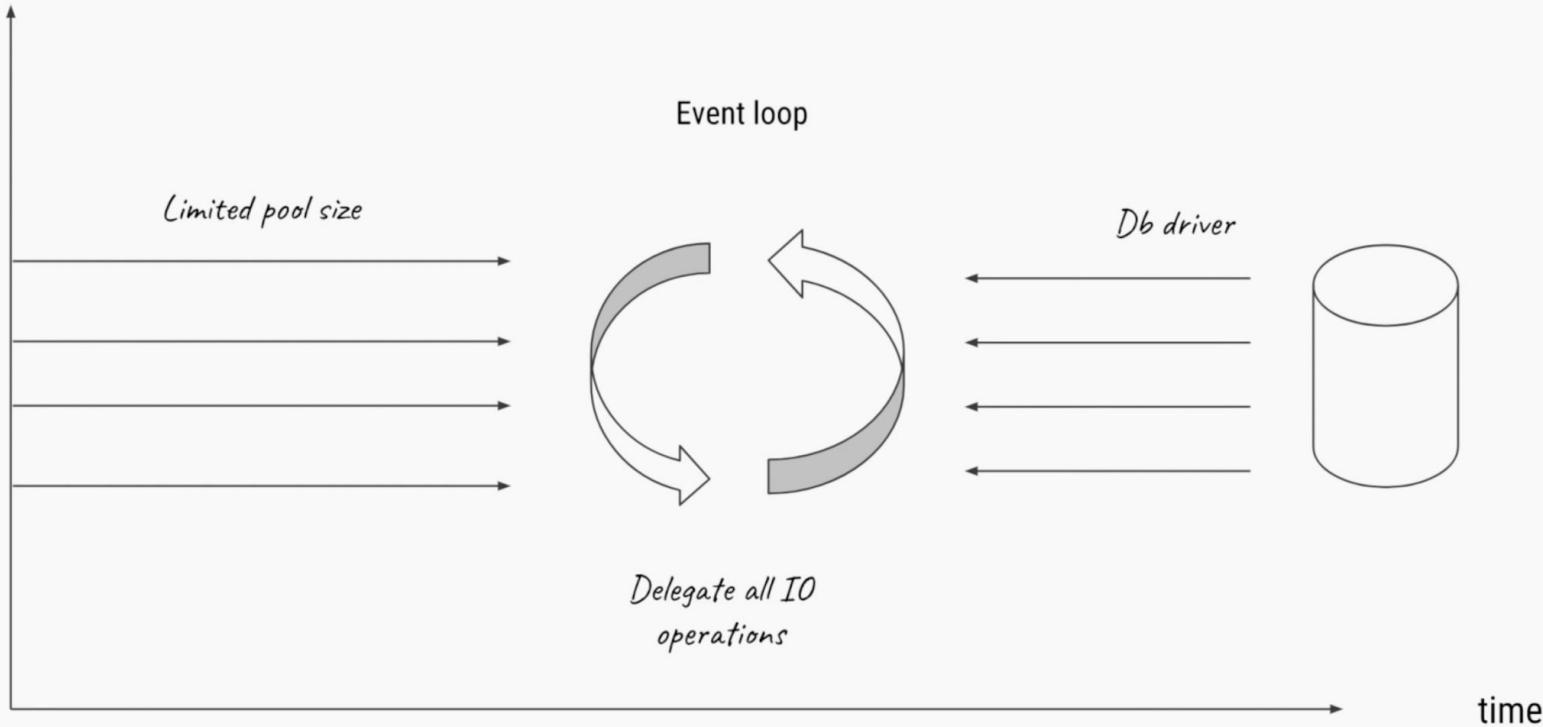
Web requests with IO

threads



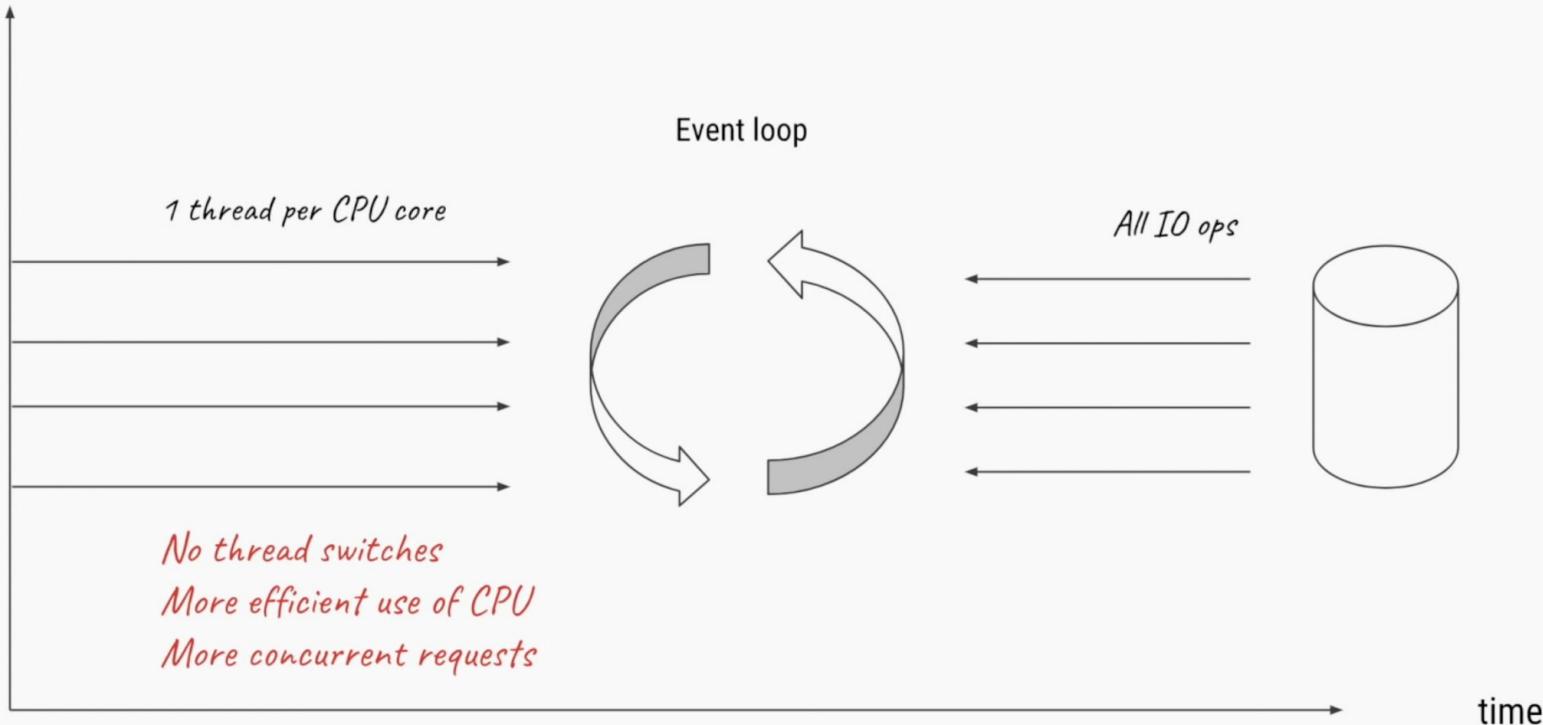
High throughput

threads



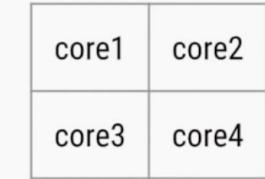
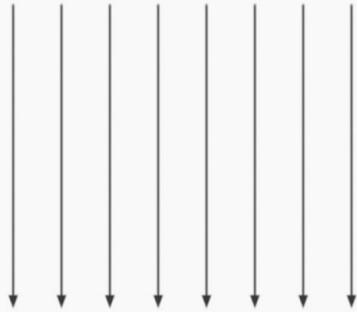
High throughput

threads



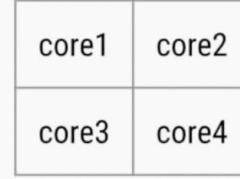
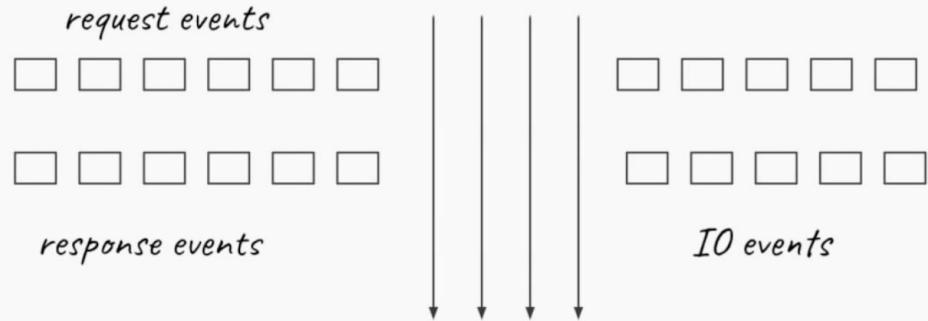
Traditional vs Event Loop

1 thread per request flow



Lot of thread switches
Scheduling of threads

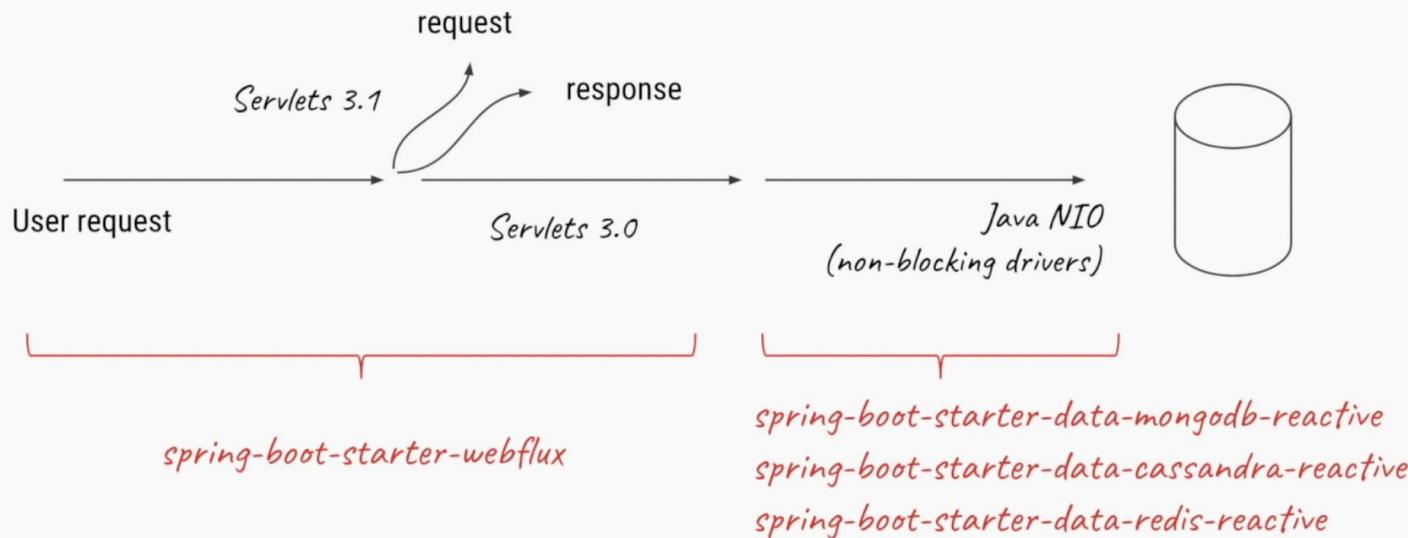
1 thread per CPU core



No thread switching
Increased complexity

End to end reactive

- Servlets 3.0 and 3.1 (Async and NIO)
- Java NIO (File, Network IO)
- MongoDB, Cassandra, Redis (more to come)



Code Perspective

Simple web request flow

```
@RestController  
public class MyRestController {  
    •  
    @Autowired  
    private UserRepository userrepository;  
  
    @GetMapping("/get/user/{id}")  
    public User getUser(@PathVariable String id) {  
        return userrepository.findUser(id);  
    }  
}
```

Blocking call

Future as placeholders?

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Future<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

CompletableFuture as placeholders?

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public CompletableFuture<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

Spring Webflux - using Reactor

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Mono<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

Mono = 0..1 elements

Flux = 0..N elements

Continuous data responses

```
@RestController
public class PricingController {

    @Autowired
    private PricesRepository pricesRepository; Reactive DB driver

    @GetMapping("/get/prices/{id}")
    public Mono<Price> getPrice(@PathVariable String id) {
        return pricesRepository.findById(id); Single value
    }

    @GetMapping("/get/prices/all")
    public Flux<Price> getAllPrice() {
        return pricesRepository.findAll(); List of values
    }

    @GetMapping(value = "/get/prices/live", produces = "text/event-stream")
    public Flux<Price> getLivePrice() {
        return pricesRepository.findAll(); Live list of values
    }
}
```

Continuous data responses

