

# Back to the Future: Strategies for Dealing with Date and Time in Test and Production Code

Marek  
Dominiak



**Marek Dominiak**

I'm running training in EventStorming,  
Architecture, DDD, and TDD

17 years of professional experience

Co-owner and instructor at



Hands-on Architect  
and Team Lead at



CTO at



# Agenda:

- Date and time in production code
- Date and time in test code
- Testing complex scenarios (Sagas)
- Ensuring rules in the codebase
- Useful tools
- Q/A

Who has experienced issues  
with date and time in production  
and / or test code?

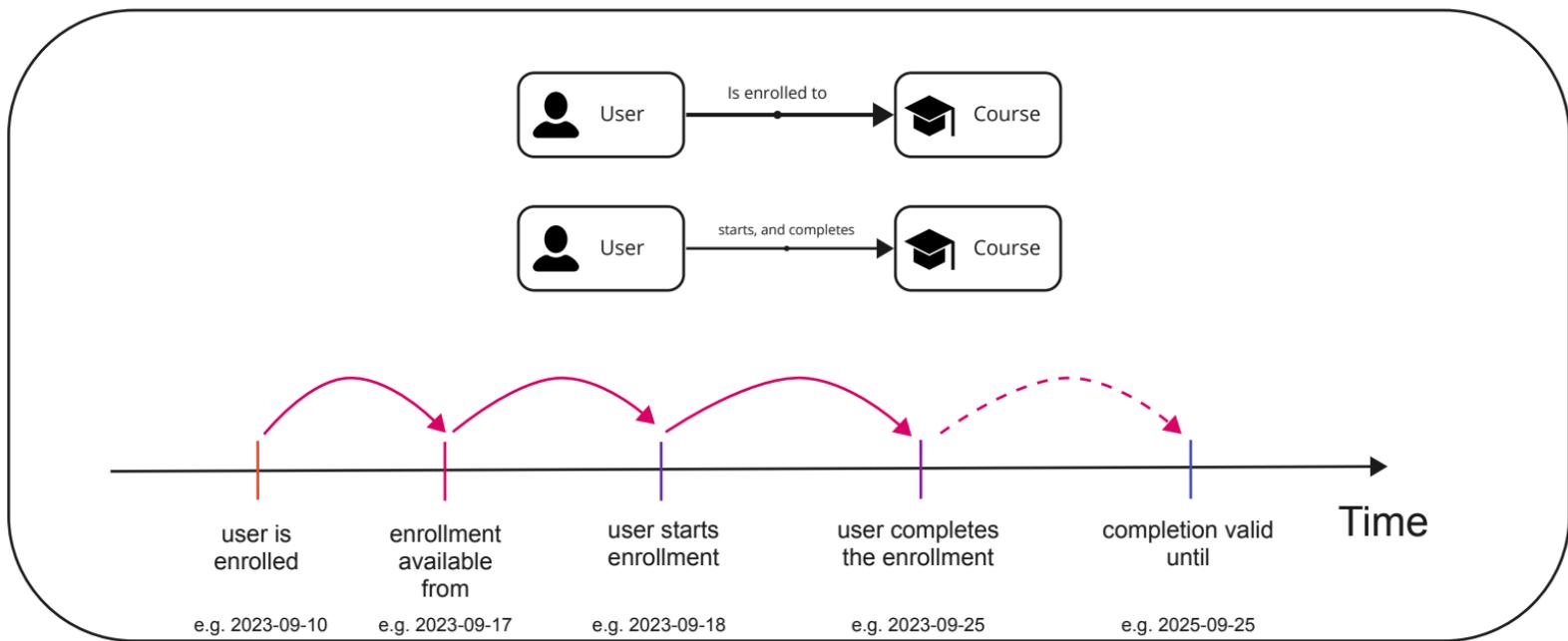
How do you create dates in our code?  
Which one do you use?

1. `new Date()`
2. `LocalDateTime.now()`
3. `new Timestamp(System.currentTimeMillis())`
4. `Instant.now()`

# Or maybe those?

1. `LocalDateTime.now(ZoneId.of("UTC"))`
2. `new Timestamp(System.currentTimeMillis())`
3. `Instant.now(Clock.systemUTC())`
4. `Clock clock = Clock.systemDefaultZone();  
clock.instant()`
5. `Clock clock = Clock.system(ZoneId.of("UTC"));  
clock.instant();`

## Example: Course Progress / Enrollment Sub-domain



# What is the difference between runs of the code on the left and right side?

```
var student = new User();
var enroller = new User();
Enrollment enrollment =
    Enrollment.builder()
        .enroller(enroller)
        .student(student)
        .course(someCourse())
        .availableFrom("2023-09-25")
        .build();
enrollment.start();
```

VS

```
var student = new User();
var enroller = new User();
Enrollment enrollment =
    Enrollment.builder()
        .enroller(enroller)
        .student(student)
        .course(someCourse())
        .availableFrom("2023-09-25")
        .build();
enrollment.start();
```

# What is the difference between runs of the code on the left and right side?

```
var student = new User();
var enroller = new User();
Enrollment enrollment =
    Enrollment.builder()
        .enroller(enroller)
        .student(student)
        .course(someCourse())
        .availableFrom("2023-09-25")
        .build();
enrollment.start();
```

VS

```
var student = new User();
var enroller = new User();
Enrollment enrollment =
    Enrollment.builder()
        .enroller(enroller)
        .student(student)
        .course(someCourse())
        .availableFrom("2023-09-25")
        .build();
enrollment.start();
```

They run at different time

# enrollment.start()

```
public class Enrollment ... {
    // ...
    public void start() {
        var now = Instant.now();
        if (now.isBefore(this.availableFrom)) {
            throw new IllegalStateException(
                "Cannot start the enrollment. Current date %s
                 is before available date %s."
                .formatted(now, this.availableFrom)
            );
        }
        this.startedAt = now;
    }
}
```

# enrollment.start()

Looks  
innocent

```
public class Enrollment ... {  
    // ...  
    public void start() {  
        var now = Instant.now();  
        if (now.isBefore(this.availableFrom)) {  
            throw new IllegalStateException(  
                "Cannot start the enrollment. Current date %s  
                is before available date %s."  
                .formatted(now, this.availableFrom)  
            );  
        }  
        this.startedAt = now;  
    }  
}
```

# enrollment.start()

Looks  
innocent

```
public class Enrollment ... {  
    // ...  
    public void start() {  
        var now = Instant.now();  
        if (now.isBefore(this.availableFrom)) {  
            throw new IllegalStateException(  
                "Cannot start the enrollment. Current date %s  
                is before available date %s."  
                .formatted(now, this.availableFrom)  
            );  
        }  
        this.startedAt = now;  
    }  
}
```

But it's  
not

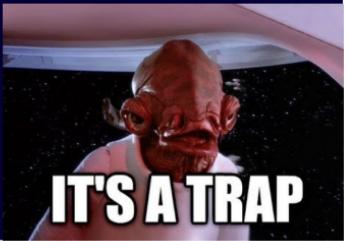
# enrollment.start()

```
public class Enrollment ... {  
    // ...  
    public void start() {  
        var now = Instant.now();  
        if (now.isBefore(this.availableFrom)) {  
            throw new IllegalStateException(  
                "Cannot start the enrollment. Current date %s  
                is before available date %s."  
                .formatted(now, this.availableFrom)  
            );  
        }  
        this.startedAt = now;  
    }  
}
```

Pandora  
box opens  
here

Looks  
innocent

But it's  
not



# They run at different times!

```
var student = new User();
var enroller = new User();
Enrollment enrollment =
    Enrollment.builder()
        .enroller(enroller)
        .student(student)
        .course(someCourse())
        .availableFrom("2023-09-25")
        .build();
enrollment.start();
```

VS

```
var student = new User();
var enroller = new User();
Enrollment enrollment =
    Enrollment.builder()
        .enroller(enroller)
        .student(student)
        .course(someCourse())
        .availableFrom("2023-09-25")
        .build();
enrollment.start();
```

Because  
run on  
2023-09-20

Throws error

They run at  
different  
time

Because  
run on  
2023-09-26

Works

# Consequences

Problem: Dependency on system clock

How can we reliably test scenarios in the past and the future if we can't control time?

## Problem: Dependency on system clock

"Solution" 1:



Let's hack our domain objects!

```
DomainObjectUtil.forceSetValue(enrollment, Enrollment.FieldName.enrollmentDate, fiveYearsAgo());  
//when & |then  
assertThat(specification.isSatisfiedBy(user)).isTrue();
```

## Problem: Dependency on system clock

"Solution" 2:



Let's add some sleeps.

```
private void forgiveMeForThisHack() {  
    try {  
        Thread.sleep( time: 1000 );  
    } catch (InterruptedException e) {  
    }  
}
```

## Problem: Dependency on system clock

"Solution" 2:



Let's add some sleeps.

```
private void forgiveMeForThisHack() {  
    try {  
        Thread.sleep( time: 1000 );  
    } catch (InterruptedException e) {  
    }  
}
```

```
TimeUnit.NANOSECONDS.sleep( timeout: 1 );
```

Problem: Dependency on system clock

"Solution" 3: Let's Mock!



- PowerMock
- Mockito from version 3.4.0

Problem: Dependency on system clock

Some "solutions":



- Delete this flickering test

Problem: Dependency on system clock

Some "solutions":



- Delete this flickering test
- Hack the domain object, bypass the domain rules

## Problem: Dependency on system clock

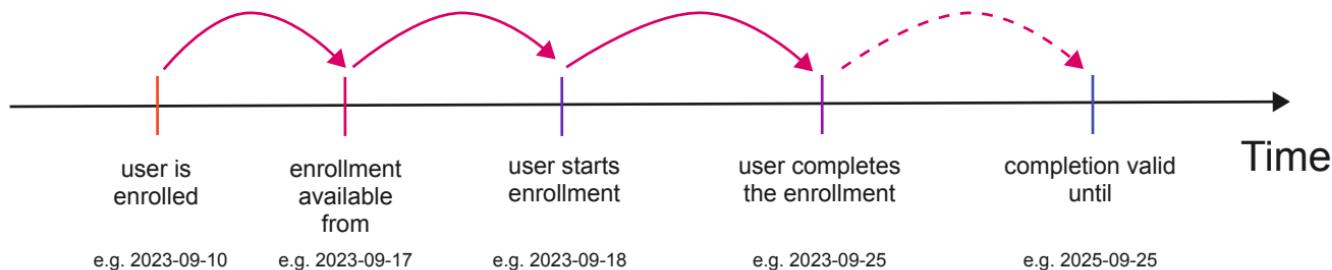
Some "solutions":



- Delete this flickering test
- Hack the domain object, bypass the domain rules
- Test manually by changing the system clock

## Some other problems

How will you test / simulate this whole process taking hours / weeks without hacking domain, and without sleeping in the code?



## Hard to test scenarios

- Online exam must take no longer than 2 hours, each question should take **minimum** 12 seconds  
=> **100 questions exam takes minimum 1200 seconds** (20 minutes)
- Notification about expiring completion is sent 2 weeks before it's become invalid (2 years - 2 weeks)?
- This list just goes on ...

## ... and more problems

- You have no idea on what to expect from the code
  - Violated The Principle of Least Surprise
- If you have to write tests for code that create dates 5 levels of code invocations below, that is no fun at all :)

... and more problems

On the outside:

```
///...
enrollment.start();
///.
```

## ... and more problems

On the outside:

```
///...
enrollment.start();
///.
```

On the inside:

```
///...
enrollment.start() calls
enrollment.doStart(...) that calls
availabilityDomainService that calls
corporateTrainingAvailPolicy that calls
some private methods
and eventually
trainingAvailabilitySpecification calls
Instant.now() /// !!!!
```

## ... and more problems

On the outside:

```
///...
enrollment.start();
///.
```

On the inside:

```
///...
enrollment.start() calls
enrollment.doStart(...) that calls
availabilityDomainService that calls
corporateTrainingAvailPolicy that calls
some private methods
and eventually
trainingAvailabilitySpecification calls
Instant.now() //!!!  
///...
```



Have fun with testing and refactoring this code

TIME is a dependency  
in our code!

## Application-wide control of TIME

- You should have control over TIME application wide.
- Meaning, there should be only one place in that service / module / project that really creates dates.

# The Clock

```
package java.time;

//...
public abstract class Clock implements
    InstantSource {
    // ...
    public abstract Instant instant();
    // ...
}
```

# Creating instants with Clock

```
// We can create them as we want
1. Clock clock = Clock.systemUTC();
   clock.instant();

2. Clock clock = Clock.system(ZoneId.of("UTC"));
   clock.instant();

3. Clock fixedClock = Clock.fixed(Instant.parse("2023
   -10-25T14:30:00.00Z"));
   clock.instant();
```

**Make the time explicit in the code!**

# Make the time explicit in the code!

```
Clock clock = ...  
User student = new User();  
User enroller = new User();  
Enrollment enrollment = Enrollment.builder()  
    .enroller(enroller)  
    .enroller(student)  
    .course(someCourse())  
    .availableFrom("2023-09-25")  
    .build();  
enrollment.startAt(clock);
```

Explicit  
parameter



# Make the time explicit in the code!

```
Clock clock = ...  
User student = new User();  
User enroller = new User();  
Enrollment enrollment = Enrollment.builder()  
    .enroller(enroller)  
    .enroller(student)  
    .course(someCourse())  
    .availableFrom("2023-09-25")  
    .build();  
enrollment.startAt(clock);
```

I know, I  
know ... It's  
a big ugly

Explicit  
parameter



# Make the time explicit in the code!

```
Clock clock = ...  
User student = new User();  
User enroller = new User();  
Enrollment enrollment = Enrollment.builder()  
    .enroller(enroller)  
    .enroller(student)  
    .course(someCourse())  
    .availableFrom("2023-09-25")  
    .build();  
enrollment.startAt(clock);
```

I know, I  
know ... It's  
a big ugly

But it is  
explicit and  
gives us  
control!

Explicit  
parameter



# Better enrollment.startAt(clock)

```
public class Enrollment {  
    // ...  
    public void startAt(Clock clock) {  
        var now = Instant.now(clock);  
        if (now.isBefore(this.availableFrom)) {  
            throw new IllegalArgumentException(  
                "Cannot start the enrollment. Current date %s  
                is before available date %s.".formatted(now  
                    , this.availableFrom)  
            );  
        }  
        this.startedAt = now;  
    }  
}
```



Explicit parameter

# Ok, so how to configure it in Spring?

```
/**  
 * Configuration for having a real Clock in the  
 * production code.  
 */  
@Profile("!test")  
@Configuration  
public class ClockConfig {  
    @Bean  
    Clock clock() {  
        return Clock.systemUTC();  
    }  
}
```

# Ok, so how to configure it in Spring?

1. Only for production code

```
/**  
 * Configuration for having a real Clock in the  
 * production code.  
 */
```

```
@Profile("!test")
```

```
@Configuration
```

```
public class ClockConfig {
```

```
    @Bean
```

```
    Clock clock() {
```

```
        return Clock.systemUTC();
```

```
    }
```

```
}
```

2.  
Singleton

# How to use it?

```
@Service  
@Transactional  
public class StartEnrollmentHandler {  
    private final EnrollmentRepository repository;  
    private final Clock clock; // 1. Inject Clock  
  
    public StartEnrollmentHandler(EnrollmentRepository repository, Clock clock) {  
        this.repository = repository;  
        this.clock = clock; // 2. Pass it to your domain object  
    }  
  
    public void startEnrollment(@NotNull UUID enrollmentId) {  
        Enrollment enrollment = repository.findById(enrollmentId)  
            .orElseThrow(() -> new IllegalArgumentException("Enrollment id=%s  
                not found".formatted(enrollmentId)));  
        enrollment.startAt(clock); // 2. Pass it to your domain object  
        repository.save(enrollment);  
    }  
}
```

# How to use it?

```
public class Enrollment {  
    // ...  
    public void startAt(Clock clock) {  
        var now = Instant.now(clock);  
        if (now.isBefore(this.availableFrom)) {  
            throw new IllegalArgumentException(  
                "Cannot start the enrollment. Current date %s  
                is before available date %s.".formatted(now  
                    , this.availableFrom)  
            );  
        }  
        this.startedAt = now;  
    }  
}
```

3. Use the Clock to get the current date

That's it!

# Look at the official docs

Best practice for applications is to pass a `Clock` into any method that requires the current instant. A dependency injection framework is one way to achieve this:

```
public class MyBean {  
    private Clock clock; // dependency inject  
    ...  
    public void process(LocalDate eventDate) {  
        if (eventDate.isBefore(LocalDate.now(clock))) {  
            ...  
        }  
    }  
}
```

This approach allows an alternate clock, such as `fixed` or `offset` to be used during testing.

# Look at the official docs

Best practice for applications is to pass a `Clock` into any method that requires the current instant. A dependency injection framework is one way to achieve this:

```
public class MyBean {  
    private Clock clock; // dependency inject  
    ...  
    public void process(LocalDate eventDate) {  
        if (eventDate.isBefore(LocalDate.now(clock))) {  
            ...  
        }  
    }  
}
```

This approach allows an alternate clock, such as `fixed` or `offset` to be used during testing.

## Important notes about Production code

- Inject Clock only in Spring Beans.

## Important notes about Production code

- Inject Clock only in Spring Beans.
- Domain objects should be given the Clock instance from the outside.

## Two types of tests

- Fast tests - no IO ops (some call them Unit tests)
- Slow tests - with IO ops (flavours of Integration tests)

## Date and time in Fast tests

- You can use `Clock.fixed(instant, zone)`
- 
- If you use Spock - you can use `MutableClock`
- 
- If you use JUnit - you can implement it on your own,  
see code of `OurMutableClock`

# In Spock: you can use Trait

```
trait UnitClockSupport {  
    final MutableClock clock = new MutableClock()  
    /**  
     * Example: adjustClock { it + ofHours(30) }, adjustClock { it - ofDays(10) }  
     */  
    MutableClock adjustClock(Closure<MutableClock> adjuster) {  
        adjuster(clock)  
    }  
    MutableClock setClockTo(Instant instant) {  
        clock.setInstant(instant)  
        clock  
    }  
    LocalDateTime localDateTime() { LocalDateTime.now(clock) }  
    LocalDate localDate() { LocalDate.now(clock) }  
    Instant instant() { Instant.now(clock) }  
}
```

Ref: <https://spockframework.org/spock/javadoc/2.0/spock/util/time/MutableClock.html>

# Example of a fast test in Spock

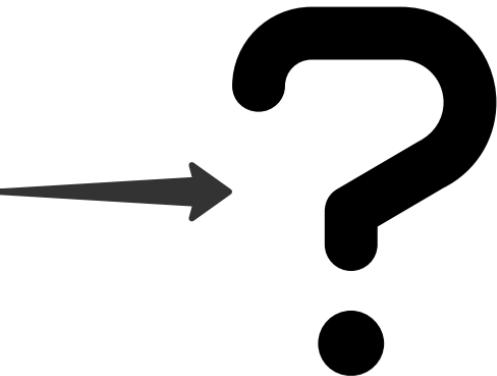
```
def "Enrollment can be started"() {  
    given:  
        def enrolledAt = Instant.now(clock)  
        def availableFrom = enrolledAt  
        Enrollment enrollment = Enrollment.initialEnrollment  
            (student, student, course, enrolledAt, availableFrom)  
  
    when:  
        adjustClock { it + ofDays(1) }  
        enrollment.startAt(clock)  
  
    then:  
        enrollment.isStarted()  
}
```

# Another example of a fast test in Spock

```
def "Enrollment can be created, started and then completed during a longer period"() {  
    given:  
    def enrolledAt = Instant.now(clock)  
    def availableFrom = enrolledAt + ofDays(14)  
    def enrollment = Enrollment.initialEnrollment(student, student, course, enrolledAt, availableFrom)  
  
    when: 'time passes'  
    adjustClock { it + ofDays(30) }  
  
    and: 'enrollment is started'  
    enrollment.startAt(clock)  
  
    and: 'some more time passes'  
    adjustClock { it + ofDays(7) }  
  
    and: 'enrollment is completed'  
    enrollment.completeAt(clock)  
  
    then:  
    enrollment.startedAt == enrolledAt + ofDays(30)  
    enrollment.completedAt == enrolledAt + ofDays(30 + 7)  
    // other verifications  
}
```

## Verification on millis/nanos

```
def "Millis verification fail"() {  
    when:  
        def now:Instant = Instant.now()  
        def now2:Instant = Instant.now()  
    then:  
        now = now2  
}
```



# Verification on millis/nanos

```
def "Millis verification fail"() {  
    when:  
    def now:Instant = Instant.now()  
    def now2:Instant = Instant.now()  
    then:  
        now = now2  
    }  
}
```

Condition not satisfied:

```
now = now2  
| | |  
| | 2023-10-21T20:31:05.202178940Z  
| false  
2023-10-21T20:31:05.201920666Z
```

Flickering  
tests

## MutableClock is awesome

It is fixed unless you adjust / change it.

```
def "Millis verification works"() {  
    when:  
        def now:Instant = Instant.now(clock)  
        def now2:Instant = Instant.now(clock)  
    then:  
        now = now2  
}
```



Test passed

# Back to the future

```
def "Back to the Future"() {  
    when:  
        setClockTo(midnightOf(LocalDate.of(1985, 10, 26)))  
        log.info("Base Time: " + localDate())  
        log.info("Marty travels to 1955\n")  
  
        setClockTo(midnightOf(LocalDate.of(1955, 11, 5)))  
        log.info("Time Travel: " + localDate())  
        log.info("Marty meets young Doc\n") // Explanation  
  
        setClockTo(midnightOf(LocalDate.of(1955, 11, 12)))  
        log.info("Enchantment Under the Sea Dance: " + localDate())  
        log.info("Marty saves his parents\n") // Explanation  
  
        setClockTo(midnightOf(LocalDate.of(2015, 10, 21)))  
        log.info("Future: " + localDate())  
        log.info("Marty and Doc arrive\n")  
  
        setClockTo(midnightOf(LocalDate.of(1885, 1, 1)))  
        log.info("Wild West: " + localDate())  
        log.info("Marty rescues Doc\n")  
        //...  
}
```



# Back to the future

```

def "Back to the Future"() {
    when:
    setClockTo(midnightOf(LocalDate.of(1985, 10, 26)))
    log.info("Base Time: " + localDate())
    log.info("Marty travels to 1955\n")

    setClockTo(midnightOf(LocalDate.of(1955, 11, 5)))
    log.info("Time Travel: " + localDate())
    log.info("Marty meets young Doc\n") // Explanation

    setClockTo(midnightOf(LocalDate.of(1955, 11, 12)))
    log.info("Enchantment Under the Sea Dance: " + localDate())
    log.info("Marty saves his parents\n") // Explanation

    setClockTo(midnightOf(LocalDate.of(2015, 10, 21)))
    log.info("Future: " + localDate())
    log.info("Marty and Doc arrive\n")

    setClockTo(midnightOf(LocalDate.of(1885, 1, 1)))
    log.info("Wild West: " + localDate())
    log.info("Marty rescues Doc\n")
    //...
}

```



## Output

BackToTheFutureSpec	: Base Time: 1985-10-26
BackToTheFutureSpec	: Marty travels to 1955
BackToTheFutureSpec	: Time Travel: 1955-11-05
BackToTheFutureSpec	: Marty meets young Doc
BackToTheFutureSpec	: Enchantment Under the Sea Dance: 1955-11-12
BackToTheFutureSpec	: Marty saves his parents
BackToTheFutureSpec	: Future: 2015-10-21
BackToTheFutureSpec	: Marty and Doc arrive
BackToTheFutureSpec	: Wild West: 1885-01-01
BackToTheFutureSpec	: Marty rescues Doc

## Date and time in Slow tests

- You can use `Clock.fixed(instant, zone)`
- If you use Spock - you can use `MutableClock`
- If you use JUnit - you can implement it on your own, see code of `OurMutableClock`
- Keep in mind that you may need to take care of your clock between tests

Ref: <https://spockframework.org/spock/javadoc/2.0/spock/util/time/MutableClock.html>

# Clock configuration for tests

```
@Configuration  
public class TestClockConfig {  
  
    @Bean  
    MutableClock clock() {  
        return new MutableClock(ZoneId.of("UTC"));  
    }  
}
```

Ref: <https://spockframework.org/spock/javadoc/2.0/spock/util/time/MutableClock.html>

# Clock configuration for tests

Located  
in test  
code

```
@Configuration
public class TestClockConfig {

    @Bean
    MutableClock clock() {
        return new MutableClock(ZoneId.of("UTC"));
    }
}
```

Ref: <https://spockframework.org/spock/javadoc/2.0/spock/util/time/MutableClock.html>

# Clock configuration for tests

Located  
in test  
code

Singleton

```
@Configuration
public class TestClockConfig {

    @Bean
    MutableClock clock() {
        return new MutableClock(ZoneId.of("UTC"));
    }
}
```

Ref: <https://spockframework.org/spock/javadoc/2.0/spock/util/time/MutableClock.html>

# Example of a slow test

```
@SpringBootTest
class StartEnrollmentHandlerTest {
    @Autowired StartEnrollmentHandler handler;
    @Autowired MutableClock clock; ← 1. Normal injections
    // ...
    @Test
    void shouldNotAllowToStartEnrollmentTooEarly() {
        // given
        var enrolledAt = date("2023-09-10");
        clock.setInstant(enrolledAt); ← 2. time manipulation
        var availableFrom = enrolledAt.plus(ofDays(1));
        var enrollment = initialEnrollment(student, student, course, enrolledAt, availableFrom);
        enrollmentRepository.save(enrollment);
        // when & then
        assertThatThrownBy(() -> handler.start(enrollment.getId()))
            .hasMessageContaining("Cannot start the enrollment"); ← 3. Method called on Spring bean
    }
}
```

# Example of a slow test

```
@SpringBootTest
class StartEnrollmentHandlerTest {
    @Autowired StartEnrollmentHandler handler;
    @Autowired MutableClock clock; // 1. Normal injections
    // ...
    @Test
    void shouldCreateASimpleEnrollmentAndStartIt() {
        // given
        var enrolledAt = date("2023-09-11");
        clock.setInstant(enrolledAt);
        var availableFrom = enrolledAt.plus(ofDays(1));
        var enrollment = initialEnrollment(student, student, course, enrolledAt, availableFrom);
        enrollmentRepository.save(enrollment);
        // when
        moveToFUTUREBy(ofDays(2));
        handler.start(enrollment.getId()); // 2. time manipulation
        // then
        assertThat(enrollmentRepository.findById(enrollment.getId()))
            .hasValueSatisfying(e -> assertThat(e.isStarted()).isTrue());
    }
}
```

2. time manipulation

3. Method called on Spring bean

# Trait for integration tests

```
@TestExecutionListeners(mergeMode = TestExecutionListeners.MergeMode.MERGE_WITH_DEFAULTS,  
                      listeners = MutableClockTestExecutionListener)  
trait IntegrationClockSupport {  
    @Autowired MutableClock clock  
  
    // Example: adjustClock { it + ofHours(30) }  
    MutableClock adjustClock(Closure<MutableClock> adjuster) {  
        adjuster(clock)  
    }  
    MutableClock setClockTo(Instant instant) {  
        clock.setInstant(instant)  
        clock  
    }  
    LocalDateTime localDateTime() { LocalDateTime.now(clock) }  
    LocalDate localDate() { LocalDate.now(clock) }  
    Instant instant() { Instant.now(clock) }  
}
```

# Back to the future Integration Test

```
@SpringBootTest
class BackToTheFutureIntegrationSpec extends Specification
    implements IntegrationClockSupport {
    @Autowired BackToTheFutureBean bean

    def "Back to the Future"() {
        when:
        setClockTo(midnightOf(LocalDate.of(1985, 10, 26)))
        bean.log("Marty travels to 1955")
        setClockTo(midnightOf(LocalDate.of(1955, 11, 5)))
        bean.log("Marty meets young Doc") // Explanation
        setClockTo(midnightOf(LocalDate.of(1955, 11, 12)))
        bean.log("Marty saves his parents") // Explanation
        setClockTo(midnightOf(LocalDate.of(2015, 10, 21)))
        bean.log("Marty and Doc arrive")
        setClockTo(midnightOf(LocalDate.of(1885, 1, 1)))
        bean.log("Marty rescues Doc")
        then:
        // ...
    }
}
```



# Back to the future Integration Test

```
@SpringBootTest
class BackToTheFutureIntegrationSpec extends Specification
    implements IntegrationClockSupport {
    @Autowired BackToTheFutureBean bean

    def "Back to the Future"() {
        when:
        setClockTo(midnightOf(LocalDate.of(1985, 10, 26)))
        bean.log("Marty travels to 1955")
        setClockTo(midnightOf(LocalDate.of(1955, 11, 5)))
        bean.log("Marty meets young Doc") // Explanation
        setClockTo(midnightOf(LocalDate.of(1955, 11, 12)))
        bean.log("Marty saves his parents") // Explanation
        setClockTo(midnightOf(LocalDate.of(2015, 10, 21)))
        bean.log("Marty and Doc arrive")
        setClockTo(midnightOf(LocalDate.of(1885, 1, 1)))
        bean.log("Marty rescues Doc")
        then:
        // ...
    }
}
```



## Output

```
(BackToTheFutureBean): Message: "Marty travels to 1955" at (1985-10-26)
(BackToTheFutureBean): Message: "Marty meets young Doc" at (1955-11-05)
(BackToTheFutureBean): Message: "Marty saves his parents" at (1955-11-12)
(BackToTheFutureBean): Message: "Marty and Doc arrive" at (2015-10-21)
(BackToTheFutureBean): Message: "Marty rescues Doc" at (1885-01-01)
```

## Slow tests: clean up before/after each test

Remember about **resetting\*** your clock:

- You can use JUnit, `@BeforeEach` with resetting the clock
- You can create a simple `@TestExecutionListener`, see:
  - `StartEnrollmentHandlerTest.java`
  - `MutableClockTestExecutionListener.java`

But how will we ensure people use Clock  
in their code?

# But how will we ensure people use Clock in their code?

```
public class DateRulesArchTest {  
    JavaClasses prodClasses = new ClassFileImporter().withImportOption(new DoNotIncludeTests  
    ()).importPackages("com.trainitek");  
  
    @Test  
    void checkThatClockIsUsedToCreateDateInProdCode() {  
        ArchRuleDefinition.noClasses()  
            .should().callMethod(LocalDate.class, "now")  
            .orShould().callMethod(LocalDate.class, "now", ZoneId.class)  
            .orShould().callMethod(LocalDateTime.class, "now")  
            .orShould().callMethod(LocalDateTime.class, "now", ZoneId.class)  
            .orShould().callMethod(ZonedDateTime.class, "now")  
            .orShould().callMethod(ZonedDateTime.class, "now", ZoneId.class)  
            .orShould().callMethod(OffsetDateTime.class, "now")  
            .orShould().callMethod(OffsetDateTime.class, "now", ZoneId.class)  
            .orShould().callMethod(Instant.class, "now")  
            .orShould().callMethod(Clock.class, "fixed", Instant.class, ZoneId.class)  
            .orShould().callMethod(Clock.class, "system", ZoneId.class)  
            .orShould().callMethod(Clock.class, "systemDefaultZone")  
            .orShould().callConstructor(Date.class)  
            .because("In prod code we use Clock to create dates (Our rules in HERE_LIK_TO_WIKI )")  
            .check(prodClasses);  
    }  
}
```

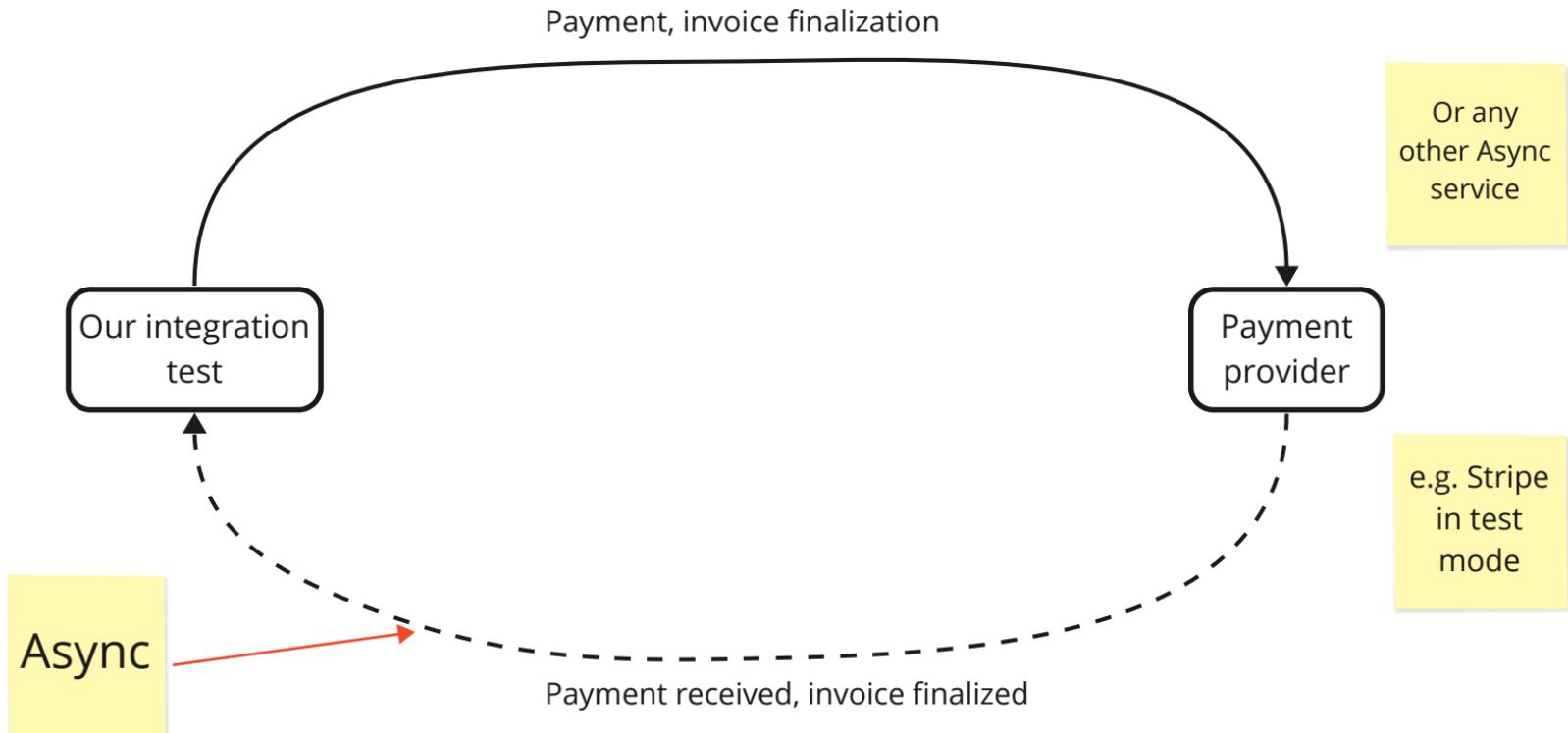
# But how will we ensure people use Clock in their code?

```
public class DateRulesArchTest {  
    JavaClasses prodClasses = new ClassFileImporter().withImportOption(new DoNotIncludeTests  
    ()).importPackages("com.trainitek");  
  
    @Test  
    void checkThatClockIsUsedToCreateDateInProdCode() {  
        ArchRuleDefinition.noClasses()  
            .should().callMethod(LocalDate.class, "now")  
            .orShould().callMethod(LocalDate.class, "now", ZoneId.class)  
            .orShould().callMethod(LocalDateTime.class, "now")  
            .orShould().callMethod(LocalDateTime.class, "now", ZoneId.class)  
            .orShould().callMethod(ZonedDateTime.class, "now")  
            .orShould().callMethod(ZonedDateTime.class, "now", ZoneId.class)  
            .orShould().callMethod(OffsetDateTime.class, "now")  
            .orShould().callMethod(OffsetDateTime.class, "now", ZoneId.class)  
            .orShould().callMethod(Instant.class, "now")  
            .orShould().callMethod(Clock.class, "fixed", Instant.class, ZoneId.class)  
            .orShould().callMethod(Clock.class, "system", ZoneId.class)  
            .orShould().callMethod(Clock.class, "systemDefaultZone")  
            .orShould().callConstructor(Date.class)  
            .because("In prod code we use Clock to create dates (Our rules in HERE_LIK_TO_WIKI )")  
            .check(prodClasses);  
    }  
}
```



## Slow Tests: Tests against other services

## Slow Tests: Tests against other services



## Slow Tests: Tests against other services

When you need to really wait until something asynchronous happens in tests use ...

## Slow Tests: Tests against other services

When you need to really wait until something asynchronous happens in tests use

- **Awaitility** for Java tests
- **PollingConditions** for Spock tests

Ref: <http://www.awaitility.org/> <https://github.com/awaitility/awaitility>

Ref 2: <https://spockframework.org/spock/javadoc/2.3/spock/util/concurrent/PollingConditions.html>

# Slow Tests: Tests against other services

## Example: Awaitility

```
private void waitAndVerifyInvoiceCanBeDownloaded(PlacedOrderDetailsResource orderDetails) {  
    Awaitility.await()  
        .atMost(Duration.ofMinutes(1))  
        .pollInterval(Duration.ofSeconds(1))  
        .untilAsserted(() -> verifyInvoiceCanBeDownloadedAndHasProperContents(  
            orderDetails.getId(), orderDetails.getCustomerName()));  
}
```

Integration  
with payment  
provider (e.g.  
Stripe)

# Slow Tests: Tests against other services

## Example: PollingConditions

Usage example:

```
def conditions = new PollingConditions(timeout: 10, initialDelay: 1.5, factor: 1.25)
def machine = new Machine()

when:
machine.start()

then:
conditions.eventually {
    assert machine.temperature >= 100
    assert machine.efficiency >= 0.9
}
```

Don't forget  
about "assert"



Ref: <https://spockframework.org/spock/javadoc/2.3/spock/util/concurrent/PollingConditions.html>

## Slow Tests: Tests against other services

Use `Awaitility` / `PollingConditions` when you have to wait for results of an async operation (that is outside our control), e.g. when integrating with:

- External services: payment providers, SMTP servers etc.
- Elasticsearch
- Messaging systems
- Anything asynchronous outside your control

TIME is a dependency!  
Take good care of it!



<https://www.linkedin.com/in/marekdominiak/>



<https://github.com/trainitek/backtothefuture>

Rate my talk in



# Q / A



<https://www.linkedin.com/in/marekdominiak/>



<https://github.com/trainitek/backtothefuture>

## Timezones are not easy

- Use timezone aware classes ONLY when you really need them.
- Timezones can be configured as user preferences (to display date and time according to their timezones)
- If you need to store only date, use LocalDate.
- Make the production app deployments more deterministic by using  
  -Duser.timezone=UTC

## More deterministic CI runs

- Ensure code on CI is parametrized with the same settings as locally:  
E.g. "-Duser.timezone=UTC" is used
- Use Clocks you can control
- Avoid midnight jumping (-/+ 1 day problem)
- Turn off unnecessary Async tasks / Cron jobs running during your tests

## Storing dates

- Use data-types offered by your datastore to save timestamps and dates:
  - Don't save year, month, day as separate columns :)
- Use UTC (UTC+0) to store all timestamps in datastores, and map them to Instant in code
- Only when you need to consider timezone use types like ZonedDateTime, OffsetDateTime

## Date formats

- Keep them as an application wide configuration
- Use specific ones as user's preferences

## "Funny" issues with date and time:

- User birthdate off by +/- 1 day.
- Production code untestable due to the passage of time.
- Threads in production code hanging indefinitely.
- Challenges debugging code due to timezone inconsistencies.
- Tests failing because of verification at millisecond precision.
- Tests failing because the machine was set up in a different timezone.
- Tests failing when run at certain times (before/after midnight).
- Flickering tests because responses from integrated services arrive too late or too early.
- Scheduled tasks running either too late or too early.
- Tests being very slow.
- Inconsistent date formats across application.
- QAs / POs waiting 14 days to check if the feature worked.