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

16 years of professional exeperience

Co-owner and instructor at trainitek

CTO at SPARKBYTE SOLUTIONS

Hands-on architect and team lead at MINTRA

Agenda:

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

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

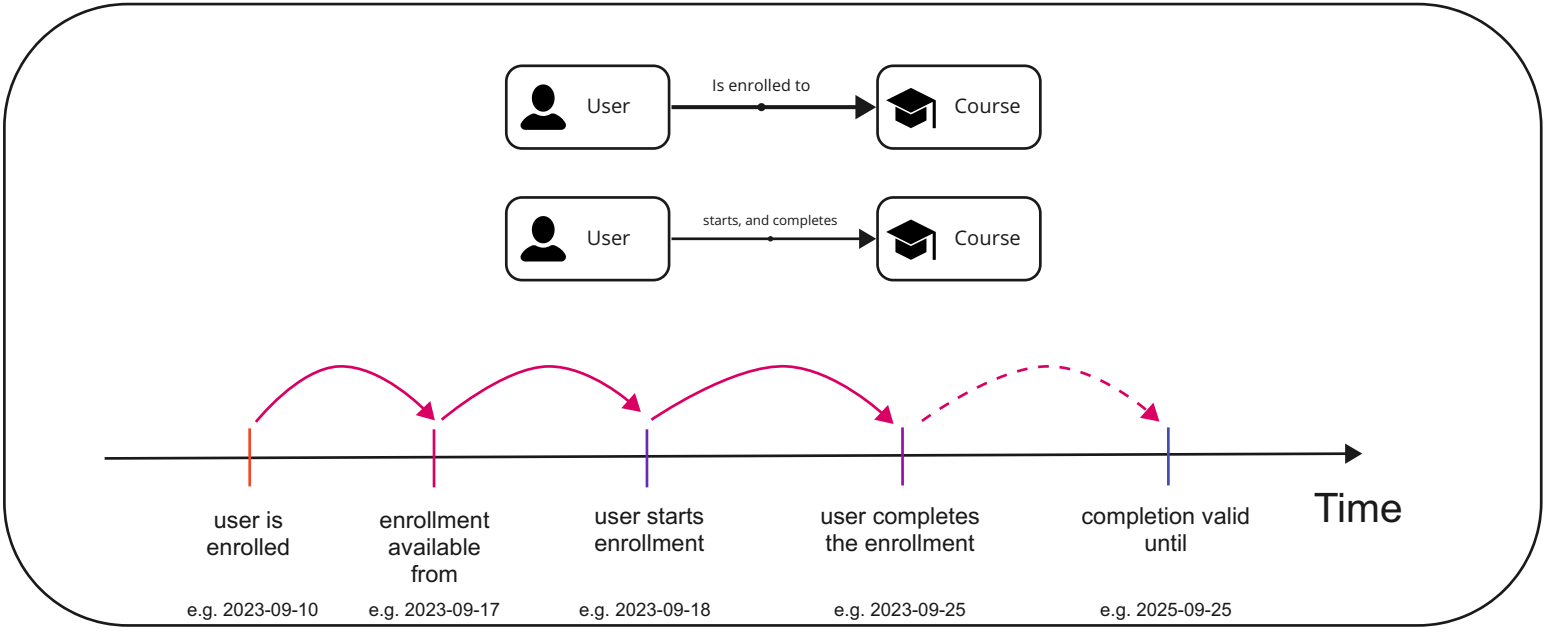
How do we create dates in our code?
Which one do you use?

```
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;
    }
}
```

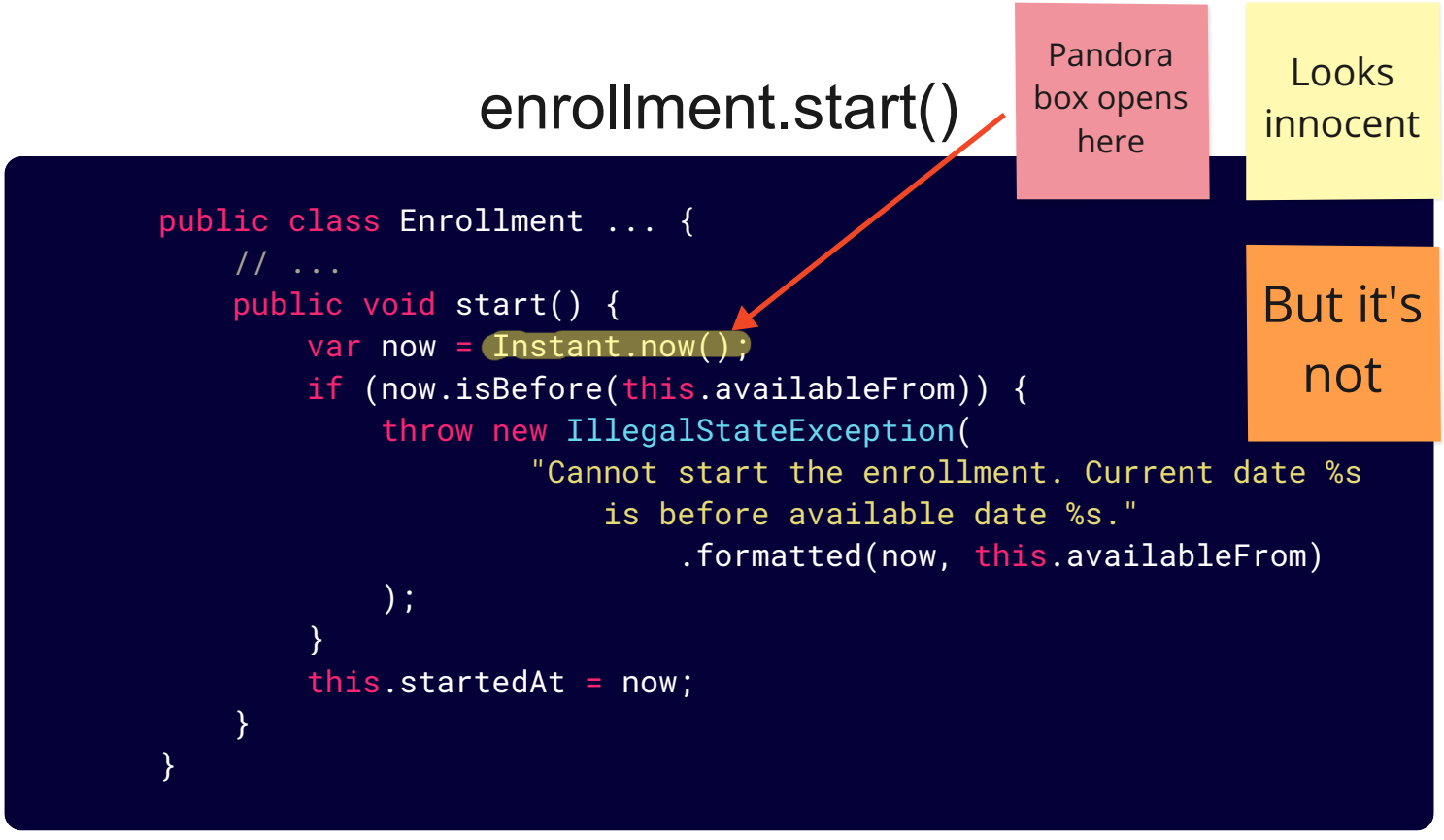
enrollment.start()

Looks innocent

But it's not

```
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()



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

Works

Because run on 2023-09-26

Consequences

Problem: Dependency on system clock

How can we reliably test scenarios in the past and 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.

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

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

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

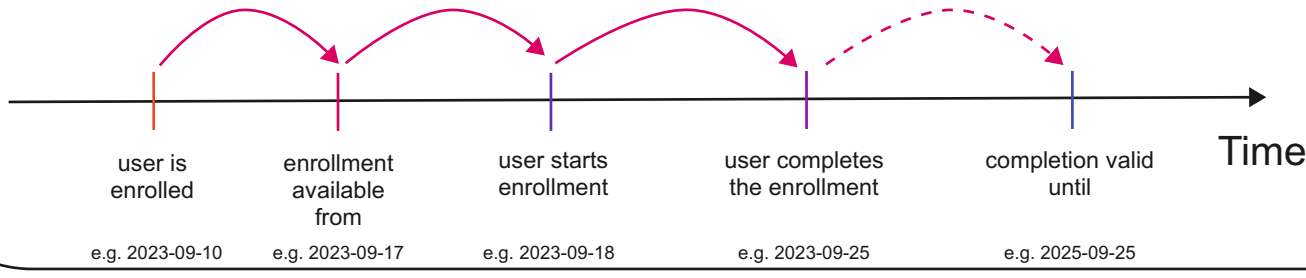
Problem: Dependency on system clock

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();
    // ...
}
```

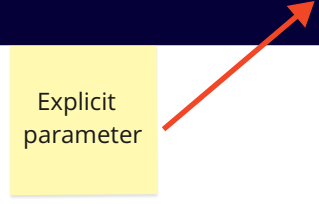
Creating instants with Clock

```
// 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);
```



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);
```

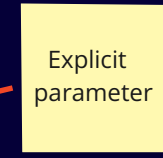
I know, I know ... It's a big ugly

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;
    }
}
```



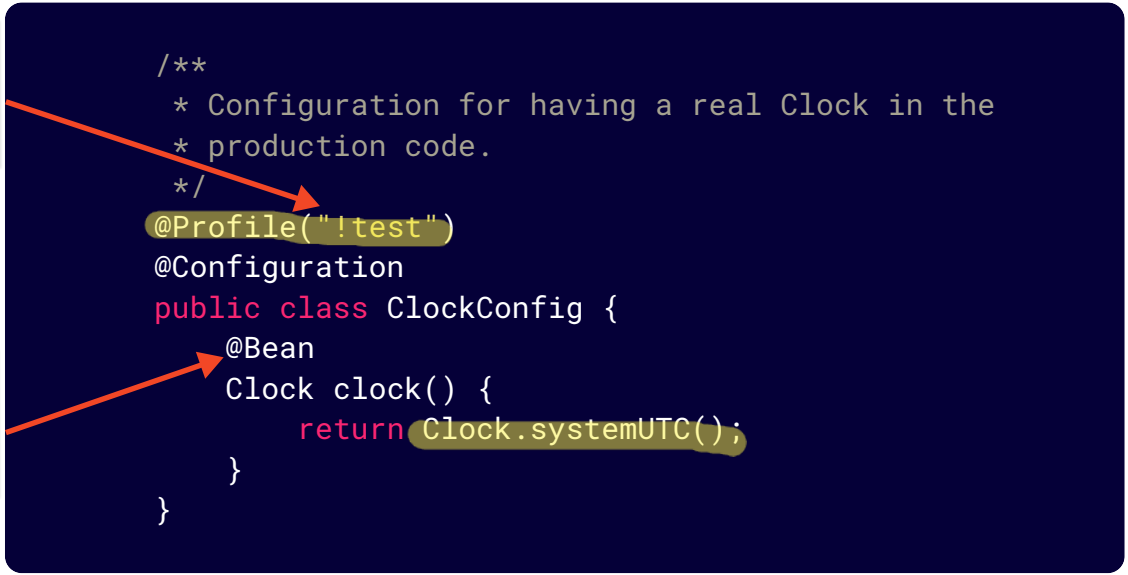
Ok, so how to configure it in Spring?

```java
/**
 * 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

2. Singleton



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

How to use it?

```java
@Service
@Transactional
public class StartEnrollmentHandler {
    private final EnrollmentRepository repository;
    private final Clock clock;

    public StartEnrollmentHandler(EnrollmentRepository repository, Clock clock) {
        this.repository = repository;
        this.clock = clock;
    }

    public void startEnrollment(@NonNull UUID enrollmentId) {
        Enrollment enrollment = repository.findById(enrollmentId)
                .orElseThrow(() -> new IllegalArgumentException("Enrollment id=%s
                    not found".formatted(enrollmentId)));
        enrollment.startAt(clock);
        repository.save(enrollment);
    }
}
```

1. Inject Clock

2. Pass it to your domain object

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.

Ref: https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html

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.

Ref: https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html

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 (it's quite simple), see code of OurMutableClock

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

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

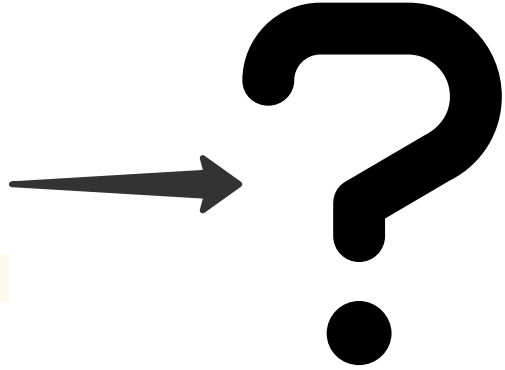
```
def "Enrollment can be started"() {
    given:
    def enrolledAt = Instant.now(clock)
    def availableFrom = enrolledAt
    Enrollment enrollment = Enrollment.initialEnrollment
        (student, student, course, enrolledAt, availableFrom)

    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 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

```groovy
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 (it's quite simple), 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

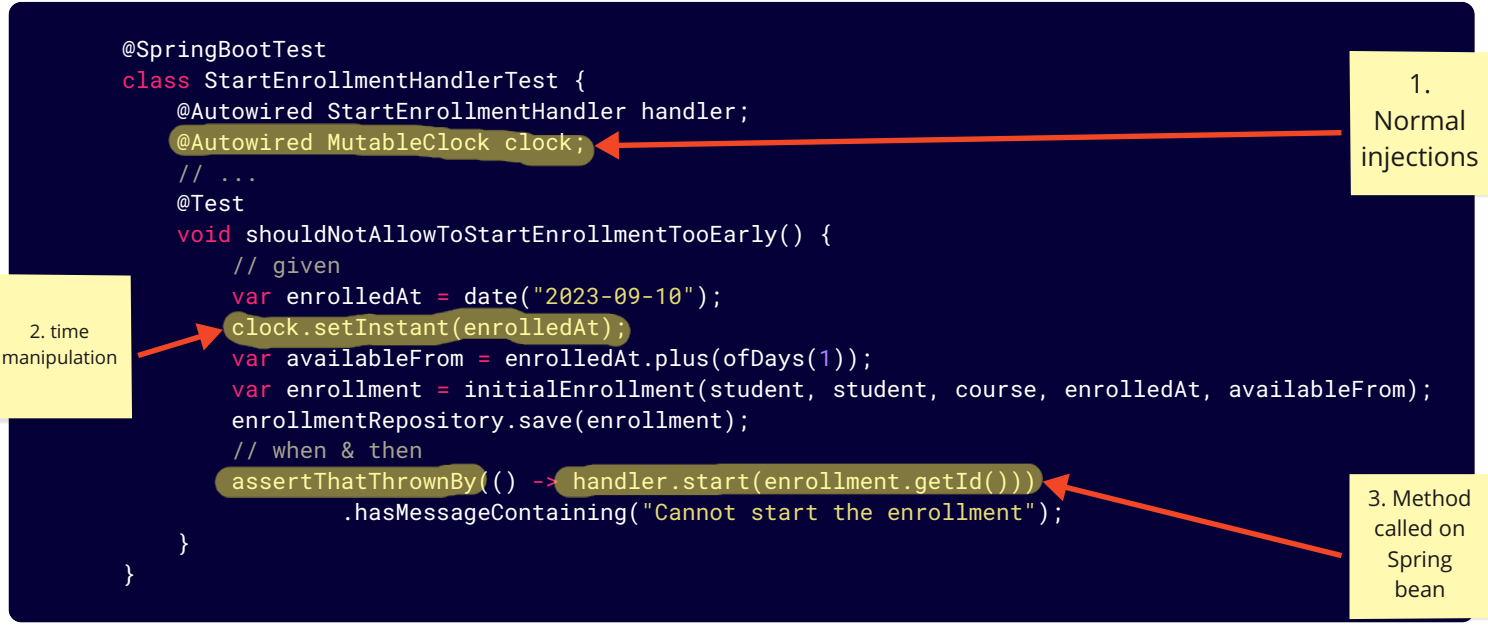
Located in test code

Singleton

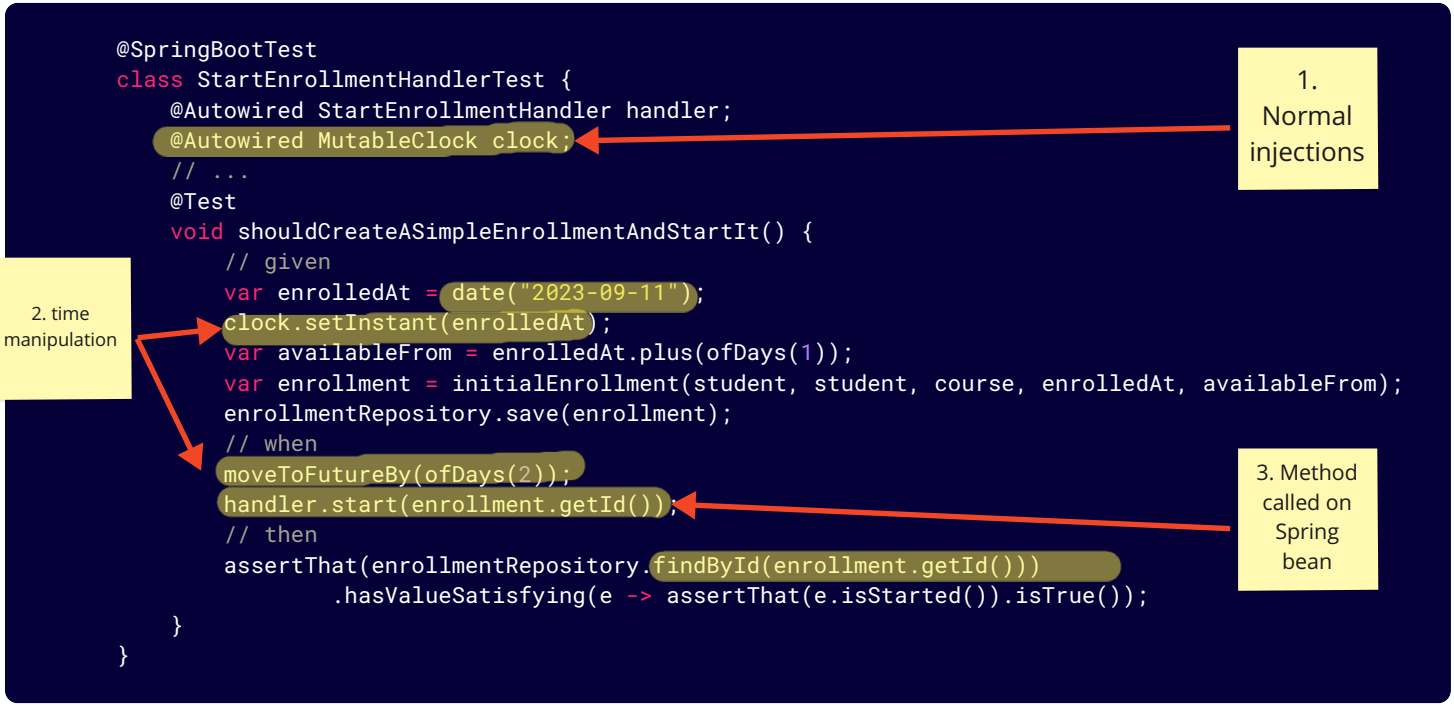
```
@Configuration
public class TestClockConfig {

    @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



Example of a slow test



Trait for integration tests

```groovy
@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

```groovy
@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

```groovy
@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);
    }
}
```

Ref: https://www.archunit.org/

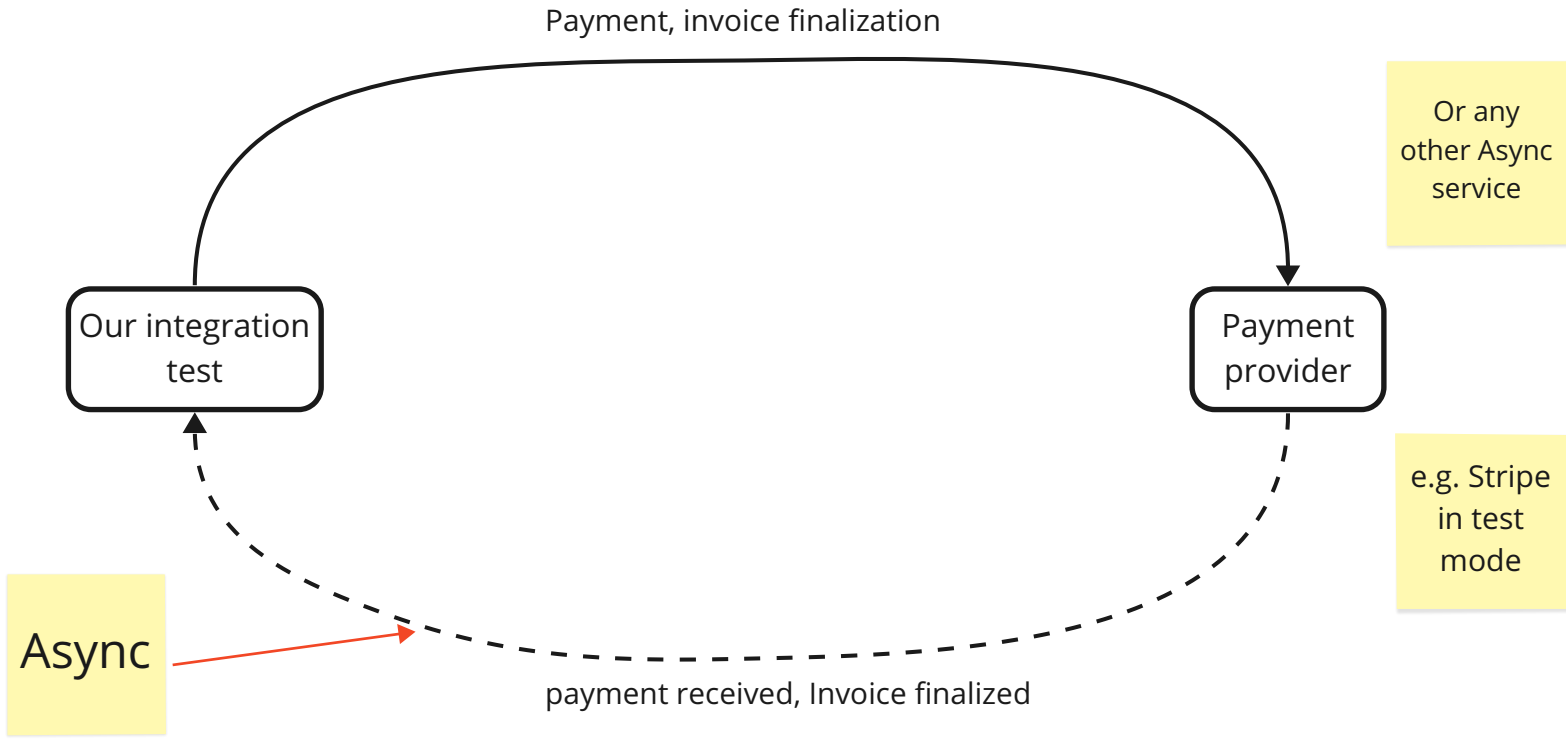
But how will we ensure people use Clock in their code?

```
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);
    }
}
```

Ref: https://www.archunit.org/

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

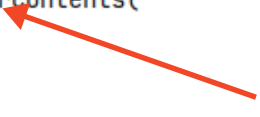
When you need to really wait until something asynchronous happens in tests use Awaitality

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

Slow Tests: Tests against other services

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

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



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

Slow Tests: Tests against other services

Use Awaitility 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

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

TIME is a dependency!
Take good care of it!

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

 https://github.com/trainitek/backtothefuture