Stop praying, start testing

Reliable Performance Testing with Gatling

The Goal of this talk

- Get you started with your first performance test
- Avoid a lot of pain down the road, by helping you write performance tests that are maintainable right off the bat
- Enable you to properly integrate your performance tests into CI
- Help you write performance tests that are actually useful
- Help you solve common problems that occur when writing performance tests

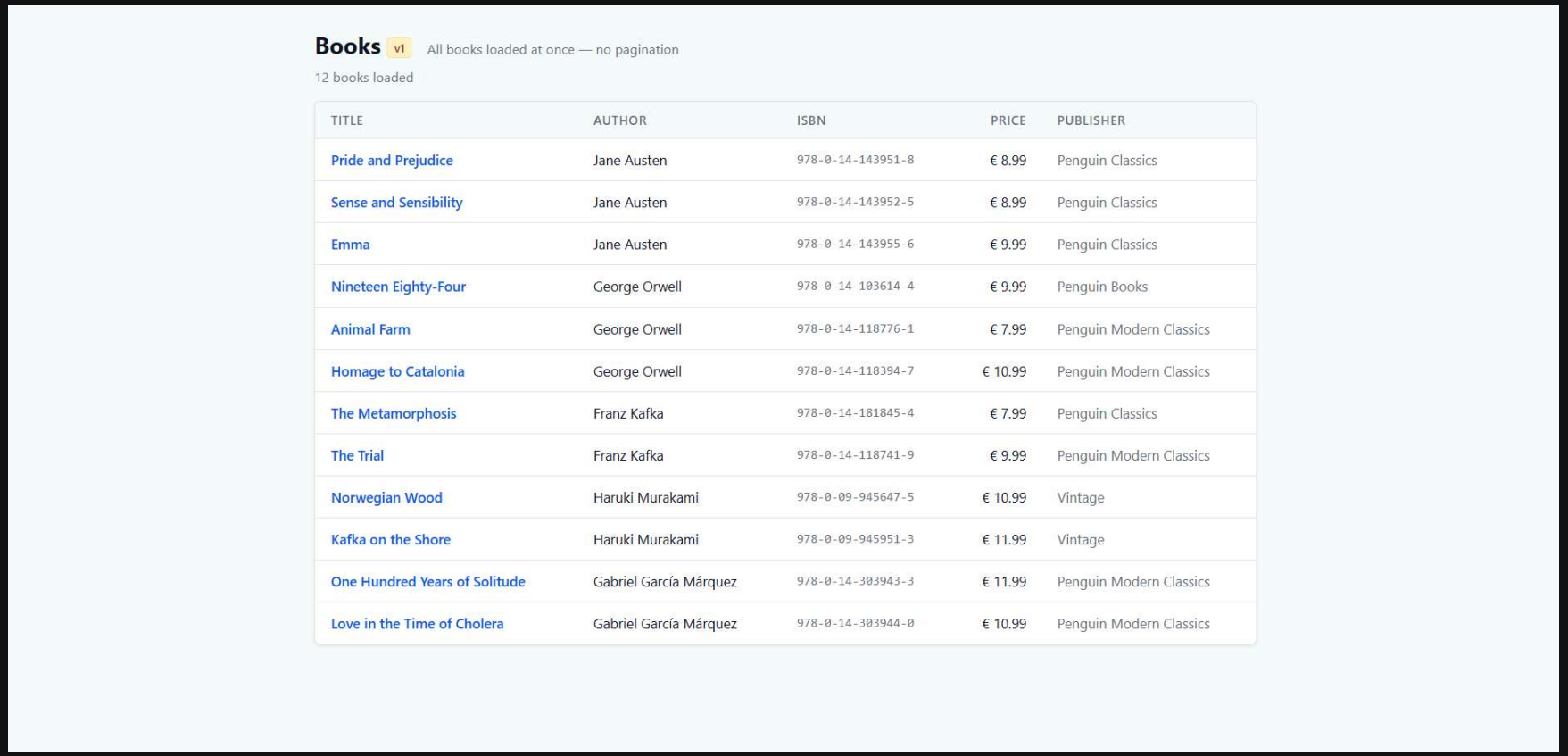
Your first performance test

Some background information on the system under test

- A tiny book store has gone viral, because of a video of an influencer
- The book store decided to create an online shop and now needs to be able to handle lots of users simultaneously
- There is a page where users can scroll through books. This feature is heavily used

Your first performance test

The UI of the application - No pagination



Books v1 All books loaded at once — no pagination

12 books loaded

TITLE	AUTHOR	ISBN	PRICE	PUBLISHER
Pride and Prejudice	Jane Austen	978-0-14-143951-8	€ 8.99	Penguin Classics
Sense and Sensibility	Jane Austen	978-0-14-143952-5	€ 8.99	Penguin Classics
Emma	Jane Austen	978-0-14-143955-6	€ 9.99	Penguin Classics
Nineteen Eighty-Four	George Orwell	978-0-14-103614-4	€ 9.99	Penguin Books
Animal Farm	George Orwell	978-0-14-118776-1	€ 7.99	Penguin Modern Classics
Homage to Catalonia	George Orwell	978-0-14-118394-7	€ 10.99	Penguin Modern Classics
The Metamorphosis	Franz Kafka	978-0-14-181845-4	€ 7.99	Penguin Classics
The Trial	Franz Kafka	978-0-14-118741-9	€ 9.99	Penguin Modern Classics
Norwegian Wood	Haruki Murakami	978-0-09-945647-5	€ 10.99	Vintage
Kafka on the Shore	Haruki Murakami	978-0-09-945951-3	€ 11.99	Vintage
One Hundred Years of Solitude	Gabriel García Márquez	978-0-14-303943-3	€ 11.99	Penguin Modern Classics
Love in the Time of Cholera	Gabriel García Márquez	978-0-14-303944-0	€ 10.99	Penguin Modern Classics

Your first performance test

An overview of the Simulation

```
object FirstSimulation : Simulation() {
    private val scenario = scenario("Browse Books V1") // Scenario: define the workflow that should be tested
    /* ... */

    fun getBooks(): ChainBuilder { /* ... */
    } // define a single request

    val HTTP_PROTOCOL = HttpDsl.http
        .baseUrl(BASE_URL) // Url to your application
        .acceptHeader("application/json")
        .contentTypeHeader("application/json")

    init {
        setUp(scenario.injectOpen(rampUsers(10_000).during(20.seconds))) // setUp: define injection profile and number
            .protocols(HTTP_PROTOCOL)
            .assertions(/* ... */)
    }
}
```

Your first performance test

Creating the scenario

```
private val scenario = scenario("Browse Books V1") // define the workflow that should be tested
    .exec(
        group("Browse Books").on( // grouping makes it easier to read the results
            exec(getBooks()), // execute the getBooks request defined below
            pause(1), // pause for one second
            exec(getBooks())
        )
    )
```

Your first performance test

Building a request

```
fun getBooks(): ChainBuilder { // define a single request
    return exec(
        http("getBooks")
            .get("${Constants.BASE_URL}/books") // GET request
            .check(status().`is`(200)) // checks the response body
    )
}
```

Your first performance test

Initializing the simulation

```
init {
    setUp(scenario.injectOpen(rampUsers(10_000).during(20.seconds))) // define injection profile and number of users
        .protocols(HTTP_PROTOCOL)
        .assertions(
            global().responseTime().percentile4() // use percentile4 instead of max to reduce flakiness of test
                .lt(100), // You need to define values according to your business requirements
            global().responseTime().mean().lt(50),
            global().successfulRequests().percent()
                .gt(95.0), // Under load some requests might fail. That can always happen
        )
}
```

Your first performance test

Running the performance test

```
./gradlew gatlingRun -your.package.FirstSimulation
```

Stats

Executions

	Total	OK	KO
Total count	702	702	0
Mean count/s	9.75	9.75	0

Response Time (ms)

	Total	OK	KO
Min	4	4	-
50th percentile	230	230	-
75th percentile	6854	6854	-
95th percentile	19433	19433	-
99th percentile	30303	30303	-
Max	37037	37037	-
Mean	4161	4161	-
Standard Deviation	6789	6789	-

Your first performance test

Analyzing the test results

- We can see that requests take too long when the load is high
- We know that there is currently no pagination
- We can derive from that, that we should probably be using pagination for the browsing of books, instead of just getting all books everytime

```
fun getBooks(page: Int? = null, size: Int? = null): ChainBuilder {
    var req = http("getBooks").get("/books")
    if (page ≠ null) req = req.queryParam("page", page) // passing request parameters
    if (size ≠ null) req = req.queryParam("size", size)
    return exec(
        req
            .check(status().`is`(200))
            .check(jsonPath("$.content").saveAs("content")) // store all response values with check().saveAs() in Gatli
            .check(jsonPath("$.totalElements").saveAs("totalElements"))
            .check(jsonPath("$.totalPages").saveAs("totalPages"))
            .check(jsonPath("$.size").saveAs("size"))
            .check(jsonPath("$.number").saveAs("number"))
```

Your first performance test

The caveats

- Great, we can now write performance tests and draw conclusions from them
- 🤔 But what if we continue adding performance tests?
- 🤔 And what if the API changes?

Making the repo maintainable

Making the repo maintainable

We should...

-  Extract Http Calls using the Page-Object pattern
-  Define response time assertions as a constant
-  Define load sizes as constants

Making the repo maintainable

Page Object Pattern

```
object BooksPage {

    fun getAllBooks(): ChainBuilder {
        return exec(
            http("getBooks")
                .get("${Constants.BASE_URL}/books") // GET request
                .check(status().`is`(200)) // checks the response body
        )
    }

    fun getBooks(page: Int? = null, size: Int? = null): ChainBuilder {
        /* ... */
    }

    /* ... */
}
```

Making the repo maintainable

Define constants

```
const val HIGH_NUMBER_OF_USERS = 10_000

val LOW_RESPONSE_TIME = arrayOf(
    CoreDsl.global().responseTime().percentile4().lt(100),
    CoreDsl.global().responseTime().mean().lt(50),
    CoreDsl.global().successfulRequests().percent().gt(95.0),
)

val MEDIUM_RESPONSE_TIME = arrayOf(
    CoreDsl.global().responseTime().percentile4().lt(1000),
    CoreDsl.global().responseTime().mean().lt(300),
    CoreDsl.global().successfulRequests().percent().gt(95.0),
)
```

- ❗ These constants change with your business requirements
- 📝 You need to define those requirements with your stakeholders
- 📝 You should consult your monitoring to figure out realistic user loads etc.

A note about the Gatling Session object

- Why do we store the response values in the session?
  - We can access them later when sending other requests
  - Storing all of the values all the time will later allow us to easily generate page objects automatically
  - But how do we access the values?
  - Read: `"#{}"` / `get` inside of exec
    - "Only Gatling SDK methods will interpolate Gatling EL Strings. You can’t use Gatling EL in your own methods or functions."
      - https://docs.gatling.io/concepts/session/el/

A note about the Gatling Session object

```
fun getBookByIdFromSession(): ChainBuilder {
    var req = http("getBookById (session)").get("/books/#{id}") // id can be accessed from session with #{id}
    return exec(
        req
            .check(status().`is`(200))
            .check(jsonPath("$.title").saveAs("title")) // store values in session for later usage
            .check(jsonPath("$.author").saveAs("author"))
            .check(jsonPath("$.isbn").saveAs("isbn"))
            .check(jsonPath("$.price").saveAs("price"))
            .check(jsonPath("$.publisher").saveAs("publisher"))
            .check(jsonPath("$.authorIds").saveAs("authorIds"))
            .check(jsonPath("$.id").saveAs("id"))
            .check(jsonPath("$.authors").saveAs("authors"))
    )
}
```

Making the repo maintainable

- This approach still creates a lot of repetitive code
  - set query params and body
  - check that the request was successful
  - save response values
  - We can avoid all of that boilerplate code using code generation

Making the repo maintainable

- How do we generate that code automatically?
- Unfortunately, there exists no good OAS Generator for page objects
- So what can we do instead?

Prompt: *Hey Claude, please create Gatling PageObjects in Kotlin and without comments for the OpenAPI Spec that I provided here:*

Making the repo maintainable

Input:

```
paths:
  /v1/books:
    get:
      tags: [ Books ]
      operationId: getAllBooks
      summary: Get all books (no pagination)
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Book'
...
```

Making the repo maintainable

Output:

```
package api

import io.gatling.javaapi.core.CoreDsl.*
import io.gatling.javaapi.http.HttpDsl.*

object BooksApi {
    val getAllBooks = exec(
        http("GET All Books - /v1/books")
            .get("/v1/books")
            .check(status().`is`(200))
    )

    val getBooksPaginated = exec(
        http("GET Books Paginated - /books")
            .get("/books")
            .queryParam("page", "#{page}")
            .queryParam("size", "#{size}")
            .check(status().`is`(200))
    )
    /* ... */
}
```

Making the repo maintainable

- 😳 This actually looks quite good
- ❌ *But:* This approach is not deterministic

A good alternative:

Prompt: *Hey Claude, write a Gradle Task that converts any OpenApi Spec into Gatling Page Objects in Kotlin without comments. Apply TDD for development. Here is a reference for the OpenAPI Spec: ...*

- ✅ A gradle task is deterministic. It always works the same
- ✅ Because the task is deterministic, it is verifiable
- ✅ If the task doesn't work, you (or AI) can fix it

I understand if you are sceptical about this approach as well. Feel free to try it out and see how well it works for you

Integrate gatling in your CI pipeline

Integrate gatling in your CI pipeline

- ✅ We now know how to write performance tests using gatling
- ✅ We know how to run performance tests and how to read the report
- ✅ We refactored the code so that changes to the API will result in the least amount of effort

But...

- 🤔 How do we integrate all of this into our pipelines?
- 🤔 When do we run tests? Periodically? On-Demand?
- 🤔 Against which stage do we run our tests?

Integrate gatling in your CI pipeline

Guidelines

- 📝 Run application and gatling **locally for debugging** purposes
- 📝 Run application and gatling **in Pipeline for CI quality gates**
- 📝 Either run application on a **dedicated server** and gatling in pipeline for **dedicated tests**
- 📝 Or run gatling against prod environment for dedicated tests (analyze when a good time is beforehand)
- 📝 Always monitor your application during performance tests

Integrate gatling in your CI pipeline

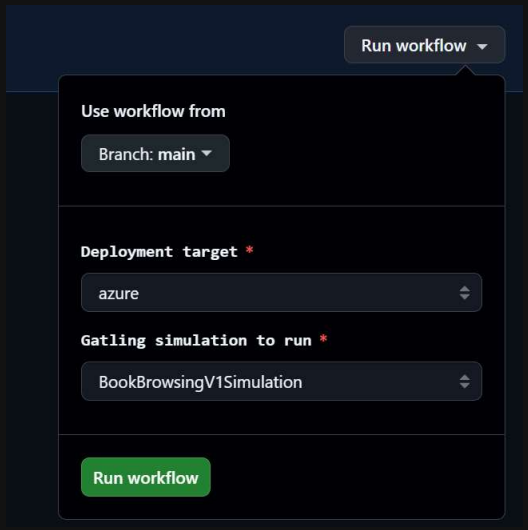
How do I run the tests in the pipeline?

- The same way you would do locally
- Run your application using docker: `docker compose up`
- Wait for the application to be ready using a custom bash probe
- Seed data if necessary. You can just create and run a dedicated simulation for that
- Run the performance test: `./gradlew gatlingRun --simulation your.package.HelloSimulation`
- Store the report and monitoring data

Integrate gatling in your CI pipeline

This is what it could look like

- Run performance tests on demand with the press of a button
- Selection for the environment against which the test should run
- Selection for the test



Integrate gatling in your CI pipeline

Deciding when to run which test

- This is pretty much on you
- You need to decide how critical the performance of specific features are for your application
- If your application is not usable with poor performance, use these tests as a quality gate for your CI
- If not, you can run them sporadically
- You should have the possibility to run a specific test against a specific stage with the press of a button and receive the gatling report and monitoring data
- A good compromise: You can define simulations that should be run during the night

How to create realistic simulations

How to create realistic simulations

- ✅ We can now create performance tests and run them in the pipeline
- 🤔 But how do we make sure that all the tests we write are useful?

Different kinds of performance tests

- Load tests: Can the system manage a certain load
- Spike tests: Can the system respond to sudden bursts of peak loads
- Stress tests: Handle peak loads that are beyond the limits of anticipated loads
- Scalability tests: How well can the system grow
- Concurrency tests: Does the system still behave correctly with many concurrent users

Translating kind of performance test into Gatling

- You need monitoring as a prerequisite
- Load profiles
  - Define an expected average load according to your monitoring data
  - Define an expected peak load according to your monitoring data
- Define acceptable response time limits for your application (business consideration)
- Define injection profiles that model your type of performance test

Gatlings injection profiles for modeling different kinds of performance tests

Profile	Description	Use Case
constantUsersPerSec	Constant rate of users per second	Sustained load testing
rampUsers	Gradually increase users over time	Realistic traffic growth
atOnceUsers	All users start at once	Spike testing
stressPeakUsers	Peak stress pattern	Find breaking points

https://docs.gatling.io/ai/assistant/vscode/create-simulation/#step-3-injection-profile

Workload Models

- There are two types of injection: closed and open
- Closed systems, where you control the concurrent number of users
- Open systems, where you control the arrival rate of users
  - https://docs.gatling.io/testing-concepts/workload-models/

- constantUsersPerSec -> closed model
- rampUsers -> open model
- The correct workload model and therefore the correct injection profile depends on the kind of test you want to create

Examples for injection profile usages

Scenario 1

- You want to test if the book store can handle the average amount of users properly
  - You start with rampUsers for a few seconds then continue with constantUsersPerSec
  - You should look into your monitoring application to figure out the average amount of users

Examples for injection profile usages

Scenario 2

- You want to test if there are memory leaks in the book store application
  - You run the constantUsersPerSec for a very long time, for example for two hours against a dedicated machine
  - You monitor memory usage and see if the memory usages increases significantly over time
  - You can do this with different sets of requests to figure out which requests cause issues
  - Once you have found significant increased memory usages you should run the application locally and use the profiler to investigate further

Examples for injection profile usages

Scenario 3

- The book store now sells tickets for signature sessions with authors and you want to make sure that the application can handle peaks when the sale starts
  - You need to guess the amount of users that try to login at peak times
  - You use atOnceUsers and see if the application can handle the load
  - In the future when you have some data about how many people tend to login at peak times you can take that number for the test

How to solve common problems

Authentication

How to test endpoints that need authentication

Goal

- Login request at the start of the simulation: `exec(Authentication.login(username, password))`
- Before every request that needs to be authenticated: `exec(Authentication.ensureValidToken)`
- Add Authorization Header to every request by adding it to the HTTP_PROTOCOL constant
- Remove Authorization Header from login and refresh calls with `header("Authorization", "")`

How to test endpoints that need authentication

Example

BrowseBooksSimulation.kt

```
scenario("Browse Books")
    .exec(Authentication.login("username", "password")) // call login at the start of the scenario
    .exec(browseBooks())
```

BooksPage.kt

```
fun getAllBooks(): ChainBuilder {
    var req = http("getAllBooks").get("$baseUrl/v1/books")
    return exec(Authentication.ensureValidToken) // call ensureValidToken before making request
        .exec(
            req
                .check(statusIs(200))
                .check(jsonPath("$[*]").ofList().saveAs("getAllBooksList"))
        )
}
```

How to test endpoints that need authentication

Implementation

```
fun login(username: String, password: String): ChainBuilder = exec(
    http("Login")
        .post(TOKEN_URL) // keycloak for example
        .header("Content-Type", "application/x-www-form-urlencoded")
        .header("Authorization", "") // authorization header should not be set here
        .formParam("grant_type", "password") // start of credentials
        .formParam("client_id", AuthenticationConfig.clientId)
        .formParam("username", username)
        .formParam("password", password) // end of credentials
        .check(
            status().`is`(200),
            jsonPath("$.access_token").saveAs("accessToken"), // store data in session
            jsonPath("$.refresh_token").saveAs("refreshToken"),
            jsonPath("$.expires_in").ofLong().saveAs("tokenExpiresIn"),
        )
).exec { session →
    session.set("tokenExpiresAt", System.currentTimeMillis() + session.getLong("tokenExpiresIn") * 1000L)
}
```

How to test endpoints that need authentication

Implementation

- Add Authorization header to every request by adding it to the HTTP_PROTOCOL constant
- Before executing an authenticated request: `exec(Authentication.ensureValidToken)`

```
val HTTP_PROTOCOL = HttpDsl.http
    .baseUrl(BASE_URL)
    .acceptHeader("application/json")
    .contentTypeHeader("application/json")
    .header("Authorization", "Bearer #{accessToken}")
```

```
val ensureValidToken: ChainBuilder =
    doIf { session →
        !session.contains("tokenExpiresAt") ||
                System.currentTimeMillis() ≥ session.getLong("tokenExpiresAt") - REFRESH_BUFFER_MS
    }.then(refreshToken)
```

How to test endpoints that need authentication

Implementation

```
val refreshToken: ChainBuilder = exec(
    http("Refresh Token")
        .post(TOKEN_URL) // keycloak for example
        .header("Content-Type", "application/x-www-form-urlencoded")
        .header("Authorization", "") // authorization header should not be set here
        .formParam("grant_type", "refresh_token")
        .formParam("client_id", AuthenticationConfig.clientId)
        .formParam("refresh_token", "#{refreshToken}")
        .check(
            status().`is`(200),
            jsonPath("$.access_token").saveAs("accessToken"),
            jsonPath("$.refresh_token").saveAs("refreshToken"),
            jsonPath("$.expires_in").ofLong().saveAs("tokenExpiresIn"),
        )
).exec { session →
    session.set("tokenExpiresAt", System.currentTimeMillis() + session.getLong("tokenExpiresIn") * 1000L)
}
```

Test Data Generation

How to generate test data

- Generating test data can become extremely painful for complex domain objects
- A good architecture for generating test data is crucial to prevent that pain
- In Gatling feeders are used to generate test data
- I really like the use of custom feeders
- You can create a custom feeder for every kind of value that exists in your application, e.g. ISBN, Names, Emails etc.
- Use Value Objects in your domain objects: Instead of storing names as Strings, create a value object Name, FirstName or LastName and store the Strings inside of that. Then write test data generation logic for each value object. Then the rest of the generation of test data can be automatated. For each field, check type and select the correct feeder

How to generate test data

Example of a feeder:

```
private val bookFeeder = intArrayOf(100).map { index →
    buildMap {
        put("title", "Title $index") // it would be nice if the values would differ more
        put("author", "Author") // it would be nice if the values were realistic and
        put("isbn", "978-11-123-${10000 + index}-21") // if the backend does validation, this needs to be a valid value
        put("price", Random.nextInt(10, 20))
        put("publisher", "Publisher")
    }
}.iterator()
```

Usage of the feeder:

```
private val scenario = scenario("Create Many Books")
    .exec(Authentication.login())
    .feed(bookFeeder)
    .exec(createBooks(NUMBER_OF_BOOKS_TO_BE_GENERATED))
```

How to generate test data

Introducing domain and value objects:

```
data class Book(
    val title: Title, // Title value object instead of String
    val author: Name, // Name value object instead of String
    val isbn: ISBN,
    val price: Price,
    val publisher: Publisher? = null,
)
```

Generating a book:

```
fun generate(): Book = Book(
    title = Title.generate(),
    author = Name.generate(),
    isbn = ISBN.generate(),
    price = Price.generate(),
    publisher = if (Random.nextBoolean()) Publisher.generate() else null,
)
```

How to generate test data

Generating a valid ISBN:

```
@JvmInline
value class ISBN(val value: String) {
    companion object {
        fun generate(): ISBN {
            val prefix = "978"
            val registrant = Random.nextInt(0, 10)
            val publication = Random.nextInt(100, 1000)
            val title = Random.nextInt(10000, 100000)
            val checkDigit = calculateCheckDigit("$prefix$registrant$publication$title")
            return ISBN("978-$registrant-$publication-$title-$checkDigit")
        }

        private fun calculateCheckDigit(digits: String): Int { /* ... */
        }
    }
}
```

How to generate test data

Generating random Names:

```
data class Name(val firstName: String, val lastName: String) {
    val fullName: String get() = "$firstName $lastName"

    companion object {
        private val FIRST_NAMES = listOf(
            "Alice", "Bob", "Clara", "David", "Elena", "Frank", "Grace",
            "Henry", "Iris", "James", "Karen", "Liam", "Maya", "Noah",
        )
        private val LAST_NAMES = listOf(
            "Ashford", "Blake", "Chen", "Drake", "Evans", "Fischer", "Grant",
            "Hayes", "Irons", "Jung", "Klein", "Lowe", "Marsh", "Nash",
        )

        fun generate(): Name = Name(FIRST_NAMES.random(), LAST_NAMES.random())
    }
}
```

How to generate test data

Mapping objects to feeders:

```
interface TestDataGenerator<T> {
    fun generate(): T
    fun toSessionMap(value: T): Map<String, Any>

    fun feeder(): Iterator<Map<String, Any>> = generateSequence { toSessionMap(generate()) }.iterator()
}
```

```
object BookFeeder : TestDataGenerator<Book> {

    override fun generate(): Book = Book.generate()

    override fun toSessionMap(value: Book): Map<String, Any> = buildMap {
        put("title", value.title.value)
        put("author", value.author.fullName)
        put("isbn", value.isbn.value)
        put("price", value.price.value)
        value.publisher?.let { put("publisher", it.value) }
    }
}
```

Summary

Summary

- You can now write different kinds of performance tests
- You know how to integrate them into your pipeline
- You know how to run them and analyze the results
- You know how to maintain the test code base
- And you know how to solve common but difficult problems that may arise during the development of tests

Feel free to connect and give feedback 😀

LinkedIn

https://www.linkedin.com/in/leon-zimmermann-5179831a4/

The github repository for this talk

https://github.com/LeonZimmermann/reliable-performance-testing