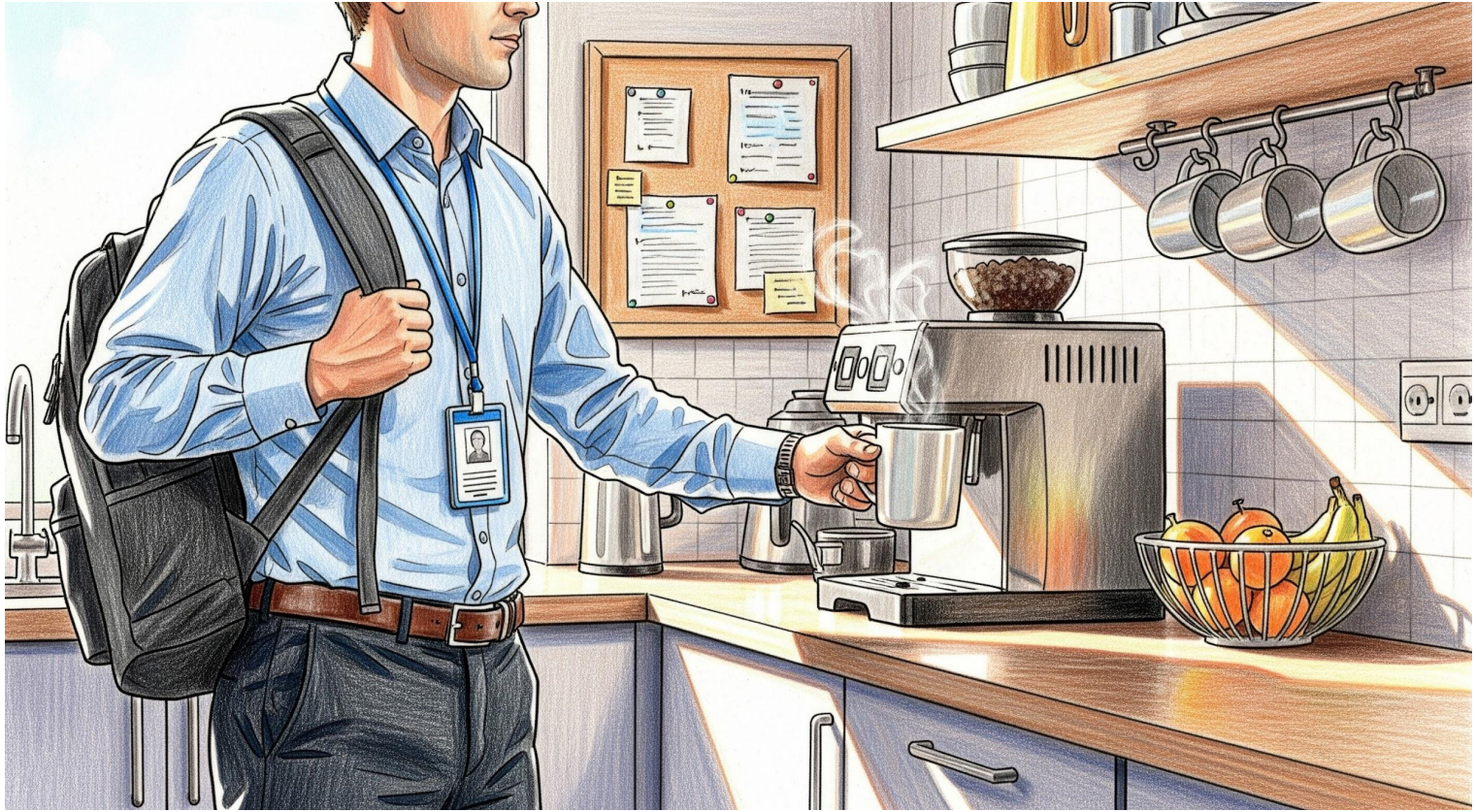
Reducing Friction with Testcontainers

in Spring Boot

Let me take you on a journey…

Has this ever happened to you?

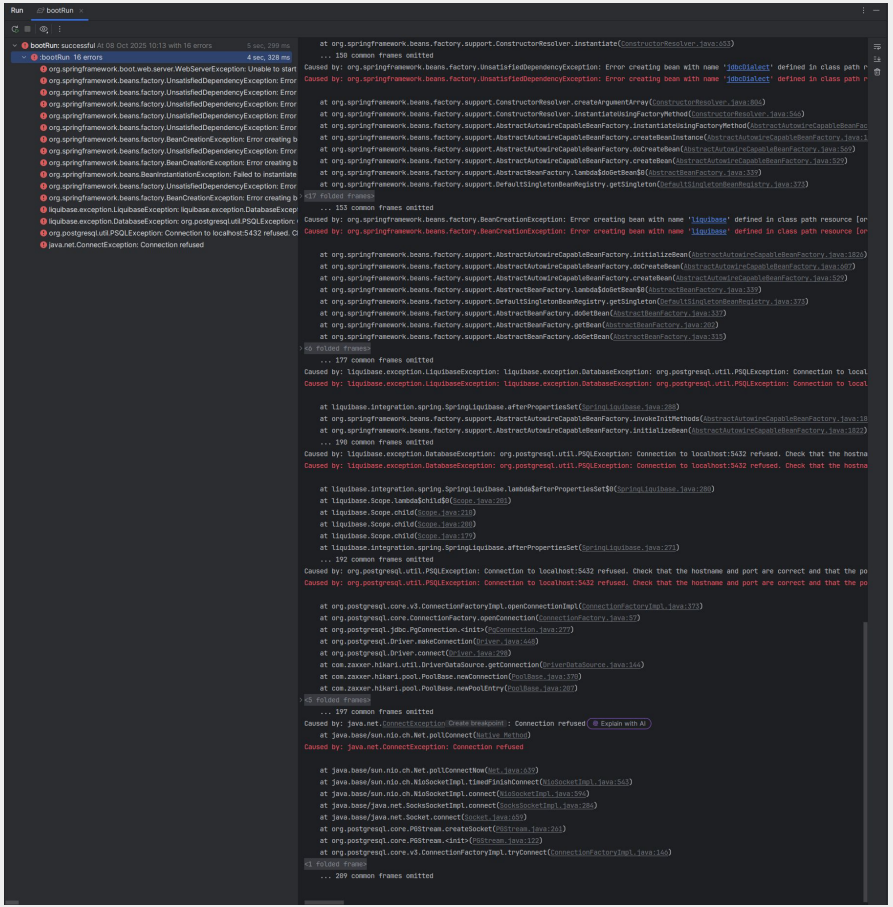


You are highly motivated and prepare to start your work day…

You sit down, focused on your starting your first task…

Errors!

Immediately after starting your project…



Damn, what is the problem?

What did I need to do again?

The Classic Way

Start your required services locally via:

- Gradle Task / Maven Goal
- Docker / Podman / other container runtime
- Systemd
- Shell Scripts
- …

This is fine in general,
but there is friction in your workflow
which is really annoying!
(Especially when switching between several projects)

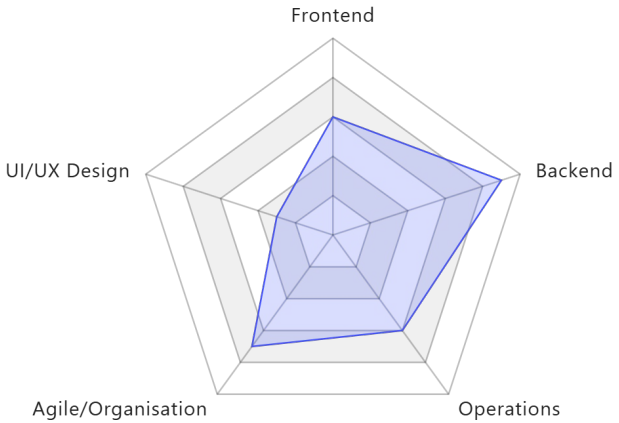
✨ A contemporary alternative! ✨

About Me

devaholics

addicted to building software

- Passionate about software development for **more than 50% of my life** now
- Gathered **over 11 years** of professional experience
- Worked in **large enterprises** as well as **startups**



Philipp Sommersguter
Co-Founder & Fullstack Developer
@ devaholics

https://devaholics.io (Web)
philipp@devaholics.io (Mail)
philipp-sommersguter (LinkedIn)

Okay, let's get the party started!

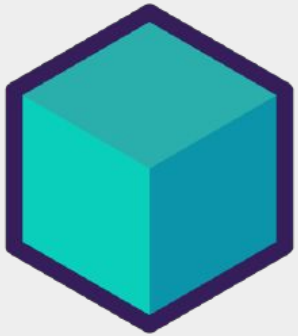
I promise, this is the last AI picture in this talk.

Agenda

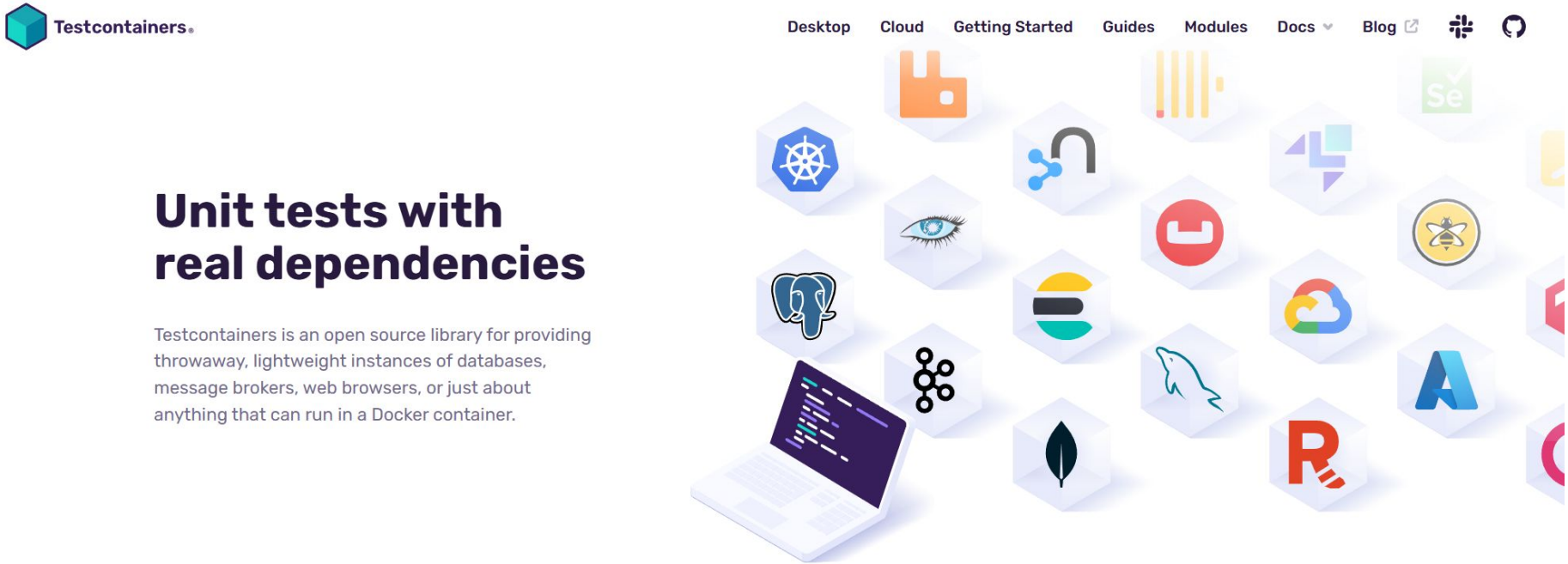
- What are Testcontainers?
- Overview of our example project
- Integration of Testcontainers into the example project
- Live Demo/Exploration
- Pitfalls (Reminder)

What are Testcontainers?

First, let's look at their website together…



Mission Statement



From their website at https://testcontainers.com/

Supported Languages

Java Go .NET Node.js Python Rust Haskell Ruby Clojure Elixir PHP Native

```
GenericContainer redis = new GenericContainer("redis:5.0.3-alpine")
    .withExposedPorts(6379);
```

Java Go .NET Node.js Python Rust Haskell Ruby Clojure Elixir PHP Native

```
RedisContainer redisContainer = new RedisBuilder().Build();
await redisContainer.StartAsync();
```

Java Go .NET Node.js Python Rust Haskell Ruby Clojure Elixir PHP Native

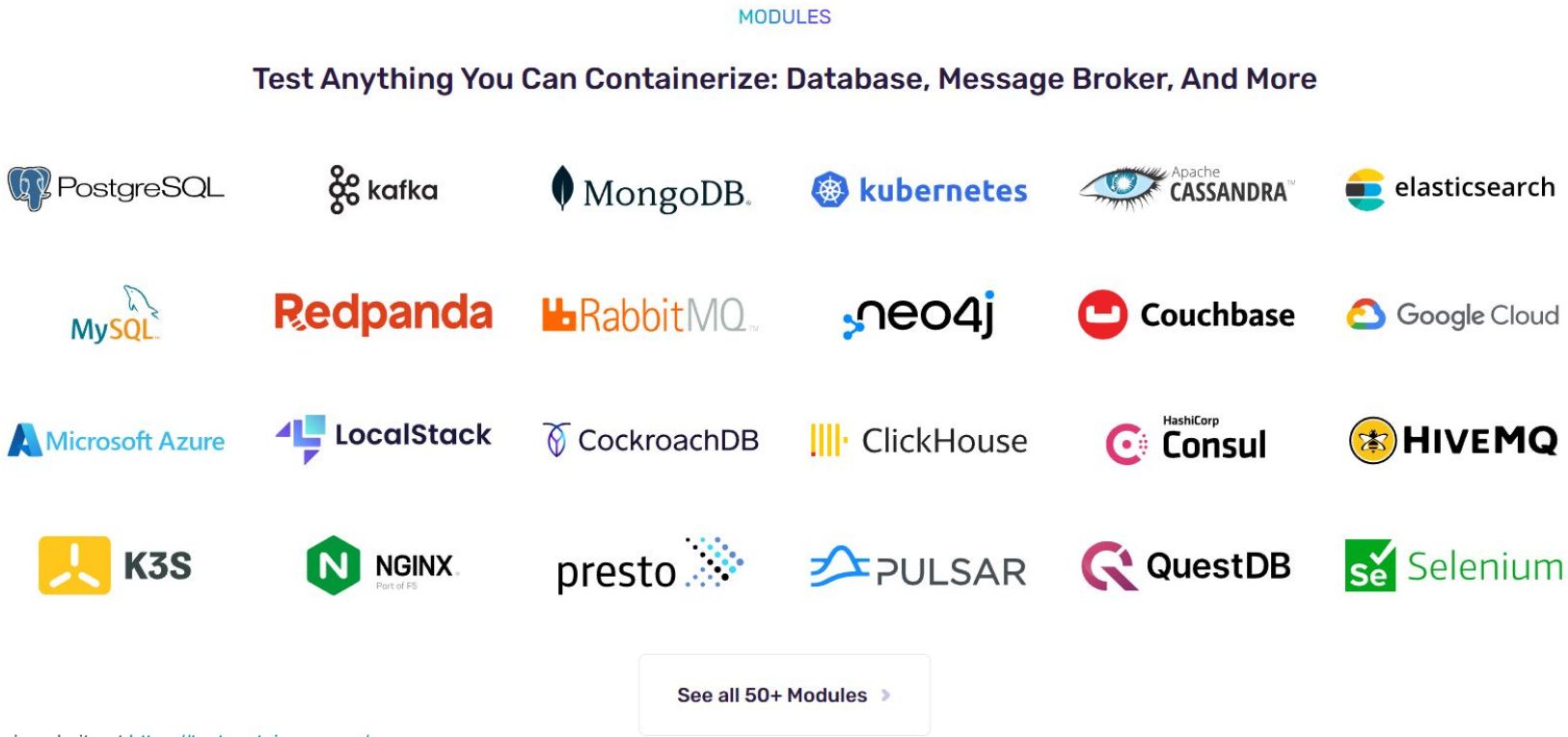
```
container, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{
    ContainerRequest: testcontainers.ContainerRequest{
        Image:        "redis:5.0.3-alpine",
        ExposedPorts: []string{"6379/tcp"},
        WaitingFor:   wait.ForLog("Ready to accept connections"),
    },
    Started:          true,
})
```

Java Go .NET Node.js Python Rust Haskell Ruby Clojure Elixir PHP Native

```
const redis = await new GenericContainer("redis:5.0.3-alpine")
    .withExposedPorts(6379)
    .withWaitStrategy(Wait.forLogMessage("Ready to accept connections"))
    .start();
```

From their website at https://testcontainers.com/

Large Collection of Ready-Made Containers (Modules)



From their website at https://testcontainers.com/

A Short History of Testcontainers

- The project started in April 2015 (first commit)
- Originally only supported running containers for unit testing
- Gained popularity starting from 2019
- People tried using it to run containers at development time soon thereafter
  - Myself included when I learned about the project in ~2019-2020
  - It wasn't really possible at that time
- Support for test containers at development shipped with Spring Boot 3.1.0 [1]

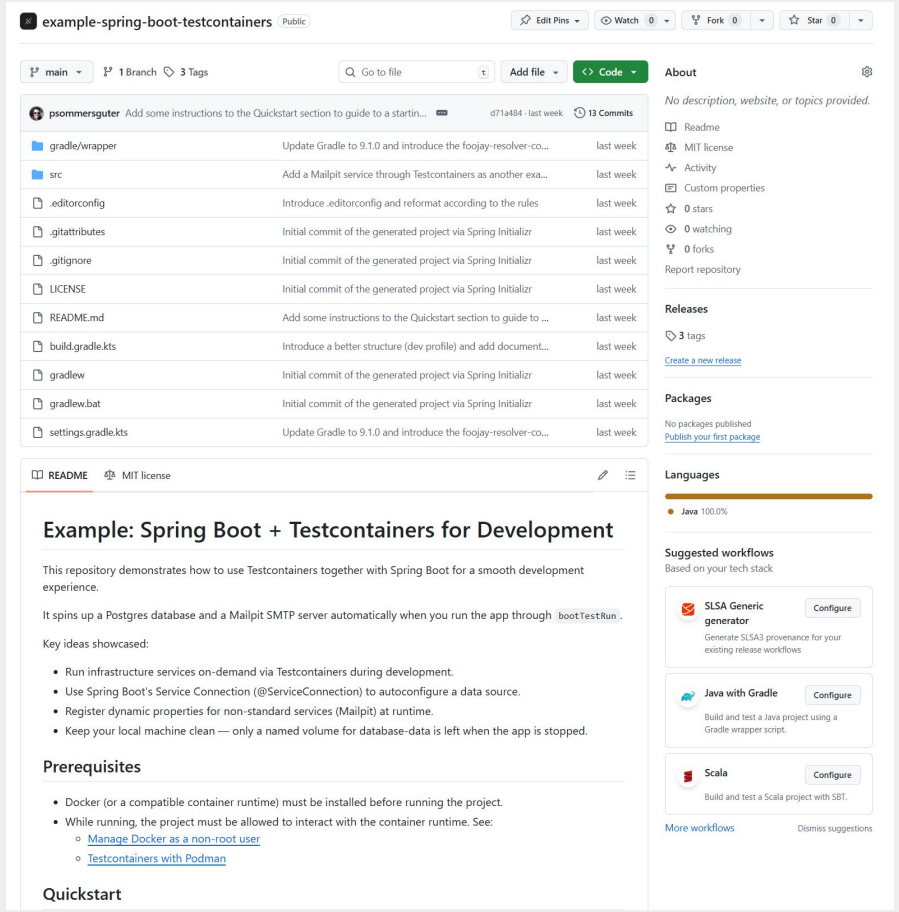
1 - https://spring.io/blog/2023/06/23/improved-testcontainers-support-in-spring-boot-3-1 (Announcement Blog Post from June 2023)

Prerequisites for Integrating Testcontainers

- You need to be familiar with containers as a technology
  - IMHO: That should generally be the case nowadays
- You (obviously) need to have a container engine installed on your machine
  - Docker
  - Podman
  - etc.
- You need intimate knowledge about the services you are using
  - Some (containerized) services have their quirks. You will see later…
  - Usually a container can be used in different ways. Know what you want and need!

Overview of the Example Project

Naturally we are building something “fancy”...



https://github.com/devaholics/example-spring-boot-testcontainers

What Should Our “Fancy Server” Be Capable Of?

1. Storing Data in a Database
    - Use PostgreSQL as a DBSM
    - Have a single database table
    - Use a very simple entity
    - On startup, generate some test entries and write them to the table
    - Read the data somehow

2. Sending Emails
    - Connect to a SMTP server
    - Send a simple test mail on demand
    - Somehow test/verify that sending the email worked

Capabilities of the Project - REST Endpoints

Provides 3 endpoints to play around with:

GET /fancy/database-content

- Dumps the database entries as lines of strings
- One line just uses the `toString()` of the entity

GET /fancy/mailpit-open

- Redirects the browser to the Mailpit Web UI
- Handy trick for development!

GET /fancy/testmail-send

- Does what the name implies

```java
@RestController  Philipp Sommersguter
@RequestMapping("/fancy")
public class SomeFancyRestController {

  private final SomeFancyService service;  4 usages

  public SomeFancyRestController(SomeFancyService service) {  Philipp Sommersguter
    this.service = service;
  }

  @GetMapping("/database-content")  Philipp Sommersguter
  public Stream<String> getDatabaseTable() { return service.streamAllAsStringLines(); }

  @GetMapping("/mailpit-open")  Philipp Sommersguter
  public ResponseEntity<String> openMailpit() {
    return ResponseEntity.status(302)
      .location(service.buildMailpitWebUiUri())
      .build();
  }

  @GetMapping("/testmail-send")  Philipp Sommersguter
  public String sendTestMail() {
    service.sendTestMail();
    return "Test mail sent successfully";
  }

}
```

Capabilities of the Project - Service

Has basic “business logic” in it but mainly glues stuff together.

- The `MailSender` and `MailProperties` beans are provided by Spring’s auto configuration
- `SomeFancyJdbcRepository` is just a default CRUD repository

```java
public interface SomeFancyJdbcRepository extends CrudRepository<SomeFancyEntity, UUID> {
}
```

```java
@Service
public class SomeFancyService {

  private final SomeFancyJdbcRepository repository;  2 usages
  private final MailSender mailSender;  2 usages
  private final MailProperties mailProperties;  3 usages

  public SomeFancyService(SomeFancyJdbcRepository repository,  Philipp Sommersguter
                          MailProperties mailProperties,
                          MailSender mailSender) {...}

  public Stream<String> streamAllAsStringLines() {  1 usage  Philipp Sommersguter
    var allEntities = repository.findAll();
    return StreamSupport.stream(allEntities.spliterator(),  parallel: true).map(SomeFancyEntity::toString);
  }

  public URI buildMailpitWebUiUri() {  1 usage  Philipp Sommersguter
    var mailpitWebPort = mailProperties.getProperties().get("mailpit.web.port");
    if (mailpitWebPort == null) {
      throw new IllegalStateException("mailpit.web.port is not set");
    }
    return URI.create("http://%s:%s/".formatted(mailProperties.getHost(), mailpitWebPort));
  }

  public void sendTestMail() {  1 usage  Philipp Sommersguter
    var now = ZonedDateTime.now();

    SimpleMailMessage mailMessage = new SimpleMailMessage();
    mailMessage.setFrom("noreply@fancy-server.com");
    mailMessage.setTo("WhomItMayConcern@whatever.com");
    mailMessage.setSubject("Testmail (%s)".formatted(now));
    mailMessage.setText("This is a testmail coming from a Spring Boot application sent at %s".formatted(now));
    mailSender.send(mailMessage);
  }
}
```

Capabilities of the Project - Database Initialization

Ensures that we have a working database:

- The function `onApplicationReady()` is called on each startup
- Ensures the table exists by executing a DDL statement through `jdbcTemplate`
- When there is no data yet, uses the `fancyJdbcRepository`[1] to create some entities with random data

- The entity[1] is pretty uninspired:

```
@Table("fancy")  3 usages  Philipp Sommersguter
public record SomeFancyEntity(@Id @Nullable UUID id, String columnA, String columnB) {
}
```

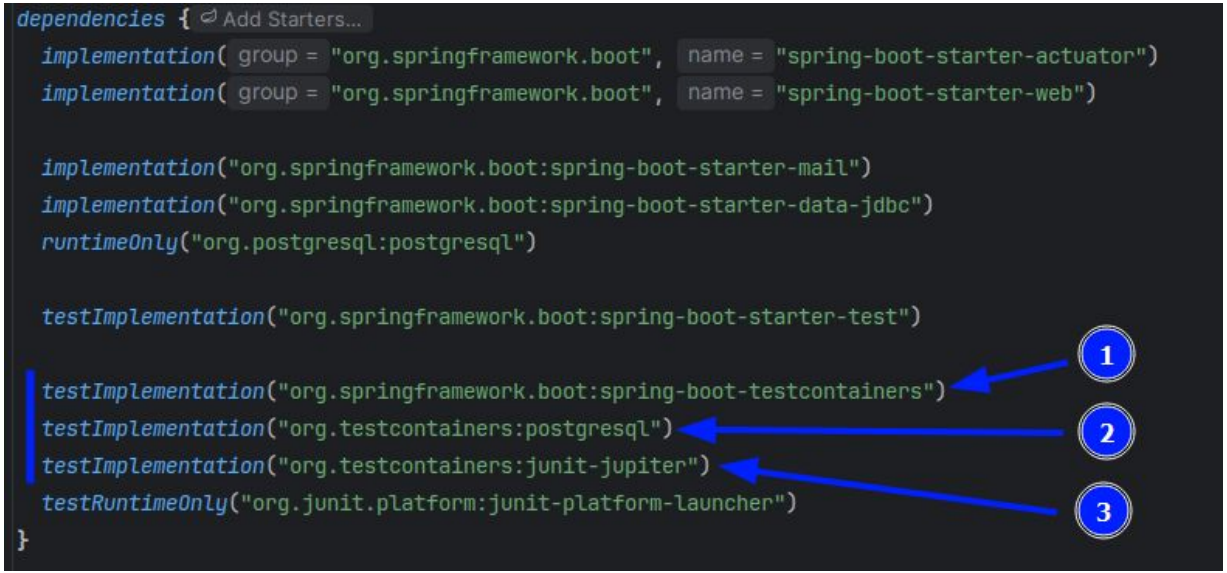
```
@Service
public class DatabaseInitializationService {

  private static final Logger log = LogManager.getLogger();  3 usages

  private final SomeFancyJdbcRepository fancyJdbcRepository;  3 usages
  private final JdbcTemplate jdbcTemplate;  2 usages

  public DatabaseInitializationService(SomeFancyJdbcRepository fancyJdbcRepository, JdbcTemplate jdbcTemplate) {...}

  @EventListener(ApplicationReadyEvent.class)  Philipp Sommersguter
  public void onApplicationReady(ApplicationReadyEvent event) {
    // Initialize database schema according to SomeFancyEntity
    jdbcTemplate.execute( sql: """
      CREATE TABLE IF NOT EXISTS fancy (
        id UUID PRIMARY KEY DEFAULT uuidv4(),
        column_a TEXT NOT NULL,
        column_b TEXT NOT NULL
      )
      """);

    // Fill the database with test data
    if (fancyJdbcRepository.count() > 0) {
      log.info("Database already filled, nothing to do");
      return;
    }

    var desiredEntriesCount = 10;
    log.info( message: "Database is empty. Filling with {} entries...", desiredEntriesCount);
    for (int i = 0; i < desiredEntriesCount; i++) {
      fancyJdbcRepository.save(new SomeFancyEntity(
          id: null,
          columnA: randomAlphabeticWord( size: 5) + "-" + i,
          randomAlphabeticWord(ThreadLocalRandom.current().nextInt( origin: 10,  bound: 20))
      ));
    }
    log.info("Database filled successfully");

  }
```

1 - The project uses spring-data-jdbc instead of spring-data-jpa with Hibernate.

Integrating Testcontainers

for usage at development time in Spring Boot

Preparation: Add the Relevant Dependencies

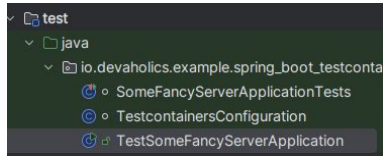


1. Enables the built-in Testcontainer support of Spring Boot
2. Provides the ready-made Postgres Module[1]
3. Integrates Testcontainers with JUnit (and therefore testing in Spring Boot)

1 - https://java.testcontainers.org/modules/databases/postgres/

Preparation: Create a Dedicated “Test” Application

We need an entrypoint in the test sources

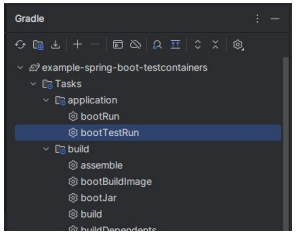


Dependencies are in the “test” scope because

a. They are rather large (~25MB build output)
b. You really don’t want that code in production!

```
public class TestSomeFancyServerApplication {  Philipp Sommersguter

  static void main(String[] args) {  Philipp Sommersguter
    SpringApplication.from(SomeFancyServerApplication::main)
      .with(TestcontainersConfiguration.class)
      .run(args);
  }

}
```

Requires us to execute the `bootTestRun` task now



Minimal Database Configuration (Postgres)

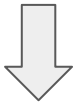
```
# Consider documentation:
# https://java.testcontainers.org/modules/databases/postgres/
# https://java.testcontainers.org/modules/databases/jdbc/#using-postgresql
spring.datasource.url=jdbc:tc:postgresql:18:///some-database
# Note: Beware of several pitfalls when using this approach:
# 1. The mapped port is dynamic and will likely change at each run.
# 2. Configured spring.datasource.username and spring.datasource.password properties will not be used.
#    Additionally, even the database name will not be used! (See https://github.com/testcontainers/testcontainers-java/issues/4121)
#    This means that the username, password and database name will be the default values. ("test")
```

In theory, super cool that you only need one property!

But, there are major pitfalls… (see comments)

Extended Database Configuration (Postgres)

Create a new `@TestConfiguration` class.



Define a bean for the container.
Noteworthy configuration:

1. Use `@ServiceConnection` for Spring to consider your configuration for the database connection
2. Fixed port to conveniently connect to the database with a database explorer tool
3. Named volume for persistence
4. Workaround for *postgres* container issue

```
@Profile("dev")  Philipp Sommersguter
@TestConfiguration(proxyBeanMethods = false)
public class DevDatabaseConfiguration {
```

```
@Bean  Philipp Sommersguter
@ServiceConnection (1)
@SuppressWarnings("resource")
PostgreSQLContainer<?> devDatabaseContainer() {
  return new PostgreSQLContainer<>( dockerImageName: "postgres:18")
    .withDatabaseName("some-database")
    .withUsername("sensible-database-username")
    .withPassword("sensible-database-password")
    .withCreateContainerCmdModifier( CreateContainerCmd cmd -> cmd
      .withName("tc-some-fancy-database")
      .withHostConfig(HostConfig.newHostConfig()
        .withAutoRemove(true) //enables faster cleanup of the container
        .withPortBindings(PortBinding.parse( serialized: "5432:5432")) (2)
        .withBinds(Bind.parse( serialized: "tc-some-fancy-database-data:/var/lib/postgresql")) (3)
      )
    )
    .waitingFor(postgresDatabaseToBeReady()); (4)
}
```

```
private static WaitStrategy postgresDatabaseToBeReady() {  1 usage  Philipp Sommersguter
  // The SQL command check works reliably and fast.
  // The only downside is that it prints a few "fatal" log messages when executing the command while the database is not created yet.
  return Wait.forSuccessfulCommand("psql -q -o /dev/null -c \"SELECT 1\" -d $POSTGRES_DB -U $POSTGRES_USER");
}
```

Email Configuration (Mailpit)

In general, do the same as before…

Noteworthy:

1. We need to use a `GenericContainer` bean as there is no official module
2. Manually set properties according to your container configuration through a `DynamicPropertyRegistrar`

But obviously, it would be too easy if that was enough…

```java
@Profile("dev")  Philipp Sommersguter
@TestConfiguration(proxyBeanMethods = false)
public class DevMailpitConfiguration {

  private static final Logger log = LogManager.getLogger();  1 usage

    A container running the Mailpit ↗ service.
  @Bean  Philipp Sommersguter
  /resource/
  GenericContainer<?> mailpitContainer() {  1
    return new GenericContainer<>( dockerImageName: "axllent/mailpit:v1.27")
      .withCreateContainerCmdModifier( CreateContainerCmd cmd -> cmd
        .withName("tc-some-fancy-mailpit")
      )
      .withExposedPorts(1025, 8025);
  }

  @Bean  Philipp Sommersguter
  DynamicPropertyRegistrar mailpitPropertiesRegistrar(GenericContainer<?> mailpitContainer) {  2
    log.info( message: "Mailpit UI is accessible through http://{}:{}", mailpitContainer.getHost(), mailpitContainer.getMappedPort( originalPort: 8025));

    return ( DynamicPropertyRegistry properties) -> {
      properties.add( name: "spring.mail.host", mailpitContainer::getHost);
      properties.add( name: "spring.mail.port", () -> mailpitContainer.getMappedPort( originalPort: 1025));
      properties.add( name: "spring.mail.properties.mailpit.web.port", () -> mailpitContainer.getMappedPort( originalPort: 8025));
    };
  }

}
```

Email Configuration (Mailpit)

Naturally, we need a small workaround…

```
spring.application.name=Some Fancy Server
spring.mail.host=dummy-to-trigger-auto-configuration_actual-value-will-be-replaced-by-property-registrar
```

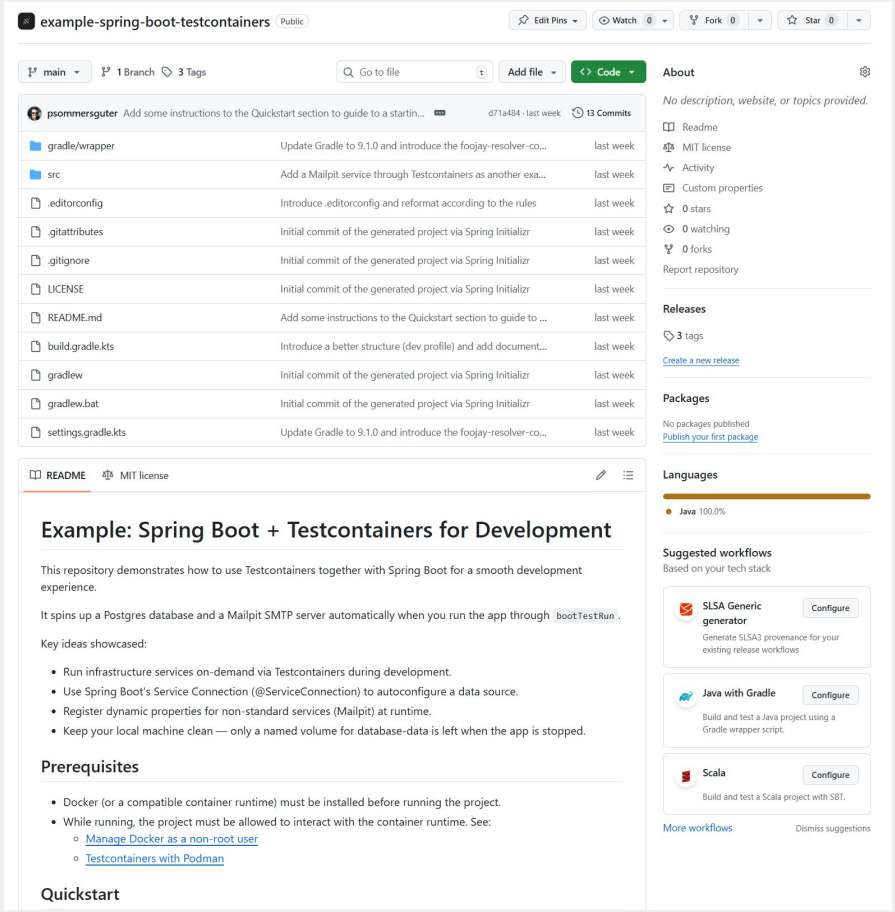
We need to set the `spring.mail.host` property since otherwise, the `MailSenderAutoConfiguration` will not be loaded.

See GitHub Issue[1] for more details.

1 - https://github.com/spring-projects/spring-boot/issues/35127

Let’s look at the whole project and try it out…

Take a look yourself!



https://github.com/devaholics/example-spring-boot-testcontainers

Notable Pitfalls (Again)

- Don't switch the import scope away from “test”
  - It will increase the bundle size ~25MB and leave yourself potentially vulnerable to nasty things.
  - Follow the "Testcontainers at development time" reference documentation[1]
- The URL property config currently is very limited for test-container databases
- Keep container specific workarounds in mind
  - Custom wait strategy for Postgres
  - Dummy property value for Mailpit
- When using a custom profile for development be aware of the extra work e.g.:
  - Run-Configurations in Gradle
  - Use `@Profile` on configuration classes
  - Use `@ActiveProfiles` on integration test classes
- Other stuff in the repository’s README[2]

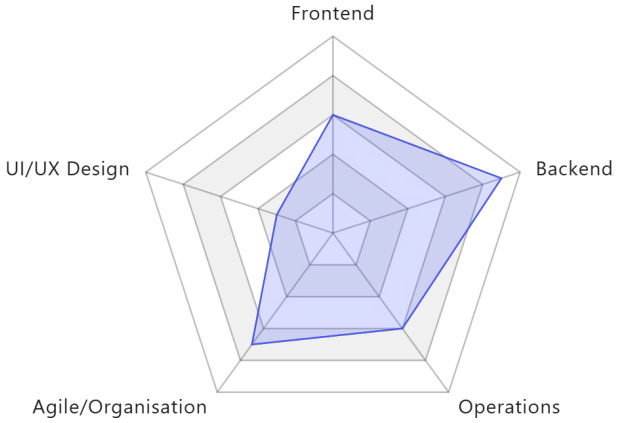
1 - https://docs.spring.io/spring-boot/reference/features/dev-services.html#features.dev-services.testcontainers.at-development-time
2 - https://github.com/devaholics/example-spring-boot-testcontainers?tab=readme-ov-file#notable-implementation-details

About Me (Again)

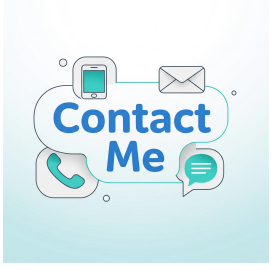
devaholics

addicted to building software

- Passionate about software development for **more than 50% of my life** now
- Gathered **over 11 years** of professional experience
- Worked in **large enterprises** as well as **startups**



Philipp Sommersguter
Co-Founder & Fullstack Developer
@ devaholics



https://devaholics.io (Web)
philipp@devaholics.io (Mail)
philipp-sommersguter (LinkedIn)

Thank you!

Any remaining questions?