





Nouveautés Spring6

David THIBAU – 2024

david.thibau@gmail.com



Agenda

Introduction

- Java17
- Spring Native
- Micrometer
- Nouveaux starters
- Modules Spring 6

Data et persistance

- Alternatives
- JPA 3.0, Hibernate 6.1
- Validation API
- Caches

Couche Web

- MVC vs Webflux. VirtualThreads
- Spring WebFlux
- RestClient et WebClient
- SSE, WebSocket

Sécurité

- What's new
- Sécurité Réactive
- Support pour OpenID/oAuth2

Spring et les Tests

- Rappels SpringTest
- Support SpringBoot
- Tests auto-configurés

Packaging et déploiement

- Jar
- Docker
- Image native



Nouvelles fonctionnalités majeures SB3.x

Java 17

Support pour la génération d'image
native

Observabilité avec Micrometer

Support de JakartaEE 10

Spring Framework 6



Améliorations

Effort pour améliorer la réactivité et la performance des applications.

- Réduire le temps de démarrage
- Réduire l'empreinte mémoire
- Optimiser l'utilisation de ressources



Compatibilité

Outils de build :

- Maven 3.5+
- Gradle 7.x

Containers de servlets :

- Tomcat 10.05.0
- Jetty 11.0 5.1
- Undertow 2.2 (Jakarta EE 9 variant) 5.0

GraalVM :

- GraalVM Community 22.3
- Native Build Tools 0.9.18



Migration

1. Mise à jour de l'infra et des pipelines pour utiliser Java 17.
2. Si utilisation de Java EE alors maj vers Jakarta EE :
 - Import **javax**. Deviennent **jakarta**.
3. Upgrader dépendances hors starters
4. Mise à jour des propriétés de configuration.
L'outil **spring-boot-properties-migrator** installé comme dépendance dans le projet peut aider.



Introduction

Java 17

Spring Native

Micrometer

Nouveaux starters

Modules Spring6



Classes scellées (JEP 409)

Les classes scellées permettent de restreindre les sous-classes d'une classe ou les implémentations d'une interface.

Seules les classes de permits peuvent hériter.

```
public sealed class Shape permits Circle, Rectangle {}  
public final class Circle extends Shape {}  
public final class Rectangle extends Shape {}
```



Interface InstantSource

Motivations :

- Abstraction du fuseau horaire fourni par *java.time.Clock*.
- Facilite la création de stubs lors des tests.

```
class AQuickTest {  
    InstantSource source;  
    ...  
    Instant getInstant() {  
        return source.instant();  
    }  
}  
...  
var quickTest = new AQuickTest(InstantSource.system());  
quickTest.getInstant();
```



Pattern Matching pour switch

```
switch (obj) {  
    case Integer i -> System.out.println("Integer: " + i);  
    case String s -> System.out.println("String: " + s);  
    default -> System.out.println("Other type");  
}
```



Nombre aléatoires

Nouvelles interfaces et implémentations pour les générateurs de nombres pseudo-aléatoires (JEP 356).

=> Plus facile d'utiliser différents algorithmes et meilleur support pour la programmation basée sur des streams

```
public IntStream getPseudoInts(String algorithm, int streamSize) {  
    // returns an IntStream with size @streamSize  
    // of random numbers generated using the @algorithm  
    // where the lower bound is 0 and the upper is 100 (exclusive)  
    return RandomGeneratorFactory.of(algorithm)  
        .create()  
        .ints(streamSize, 0, 100);  
}
```



Autres

Foreign Function et Memory API (Incubation):

Permet d'accéder au code et à la mémoire en dehors de la JVM. Remplacer JNI à termes.

Vector API (Incubation) :

Traitement vectoriel. Une instruction exécutée en // sur plusieurs données

Garbage collector :

ZGC (Garbage Collector basse latence) prêt pour la production (JEP 391).

Record :

Peuvent être utilisés dans des classes internes et être scellés.

MacOs :

Java2D => Apple Metal API
Architecture AArch64

Suppression, dépréciation :

API applet, API RMI Activation, Compilateurs AOT et JIT, Security Manager,
Moteur Javascript Nashorn

Encapsulation plus forte de JDK internals



Introduction

Java 17
Spring Native
Micrometer
Nouveaux starters
Modules Spring6



Image native

Technologie permettant de générer du code Java dans un exécutable autonome comprenant

- les classes d'application,
- les classes des dépendances
- les classes de la bibliothèque d'exécution (JVM)
- le code natif lié statiquement

=> Pas nécessaire d'avoir un environnement d'exécution Java sur le système cible

=> l'artefact de génération dépend de la plateforme.

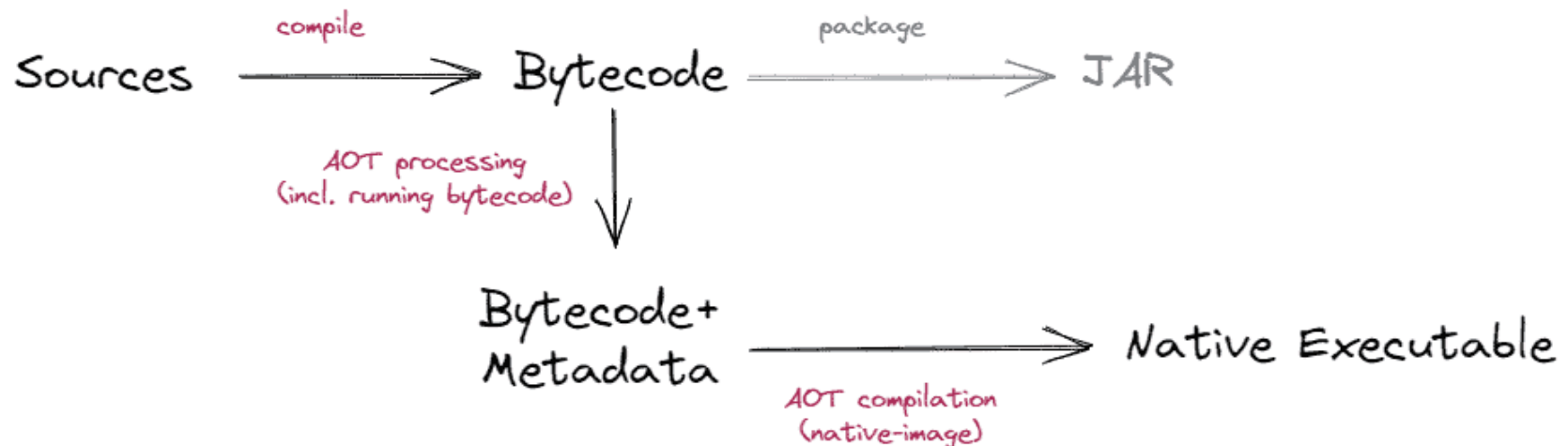
Déploiement facilité par Docker.

AOT

La compilation ***Ahead-Of-Time (AOT)*** est le processus de compilation de code Java en code exécutable natif.

Il est souvent exécuté en 2 étapes :

- AOT processing : collecte de métadonnées pour les fournir au compilateur AOT. Framework dépendant
- Compilation AOT





Closed World Optimization

Les fonctionnalités de reflexion et de scan des annotations au démarrages n'apportent rien dans un environnement cloud.

Avec les images natives, les traitements faits habituellement au runtime sont déportés durant la compilation.

=> Amélioration drastique des temps de démarrage. Réduction utilisation mémoire

=> Limitations sur le code de la JVM



Limitations

Les initialiseurs de classe sont exécutés au moment de la construction.

=> Cela peut briser certaines hypothèses dans le code.

La réflexion et les proxys dynamiques, coûteux au runtime, sont optimisés au build-time.

=> L'utilisation de proxys dynamiques doit être annoncée au compilateur AOT.

Idem pour JNI et la sérialisation.

Avec les images natives, le bytecode n'est plus disponible au moment de l'exécution, donc le débogage et la surveillance avec des outils ciblés sur le JVMTI ne sont pas possibles¹.

=> Utiliser des débogueurs et outils de surveillance natifs .



Support Spring

Les plugins Maven et Gradle

- Ont des cibles bas-niveau ***process-aot, process-test-aot***, ...
- Fournissent un profil native

La plupart des starters préviennent le compilateur lorsqu'ils utilisent la reflexion ou le chargement dynamique de ressource

Certains ne le prennent en charge c'est alors au développeur de faire ces déclarations

- Via les fichiers de méta-données du compilateur
- Programmatique
- Ou via des annotations. Ex :
@RegisterReflectionForBinding



Introduction

Java 17
Spring Native
Micrometer
Nouveaux starters
Modules Spring6



Observation API

Micrometer Observation API permet de masquer les API de bas-niveau.

L'API est basée sur l'interface ***Observation***

- => Plus besoin de penser à des abstractions de bas niveau comme un compteur de temps, d'invocation ou tracer, simplement dire ce que nous voulons observer.

–

Supporte les systèmes de monitoring classiques :
Atlas, Datadog, Graphite, Ganglia, Influx, JMX,
Prometheus.



Actuator

L'intégration de micrometer s'effectue via la dépendance **Actuator** qui auto-configue un **ObservationRegistry** et des **ObservationHandlers** collectant les métriques

```
@Service
public class GreetingService {

    private ObservationRegistry observationRegistry;

    public String sayHello() {
        return Observation
            .createNotStarted("greetingService", observationRegistry)
            .observe(this::sayHelloNoObserver);
    }

    private String sayHelloNoObserver() {
        return "Hello World!";
    }
}
```



Autres apports

Web : Auto-configuration de filtres permettant des observations http et le tracing distribué

AOP et l'annotation *@Observed*. Compte le nombre d'appels de méthodes, leur temps total d'exécution, le max

Test : *micrometer-observation-test*

Tracing : *micrometer-tracing*, définition de spans



Introduction

Java 17
Spring Native
Micrometer
Nouveaux starters
Modules Spring6



Outils de build

Choix par défaut : Gradle/Groovy

Autre choix supportés :

- Gradle/Kotlin
- Maven



Outils développeur

Support GraalVM : Images natives

GraphQL code generation : Génération de code à partir du schéma GraphQL

Docker compose support : Démarrage automatique des services d'appui

Testcontainers : Démarrage de service d'appui durant les tests

Modulith : Support pour applications modulaires



Autres

Web :

- Spring for GraphQL
- JTE : Moteur de template
- SpringDoc OpenAPI

Sécurité

- Oauth Authorization Server

Message Broker

- Streams pour Rabbit MQ

Testing

- TestContainers

AI

- APIs, Bases Vectorielles



Introduction

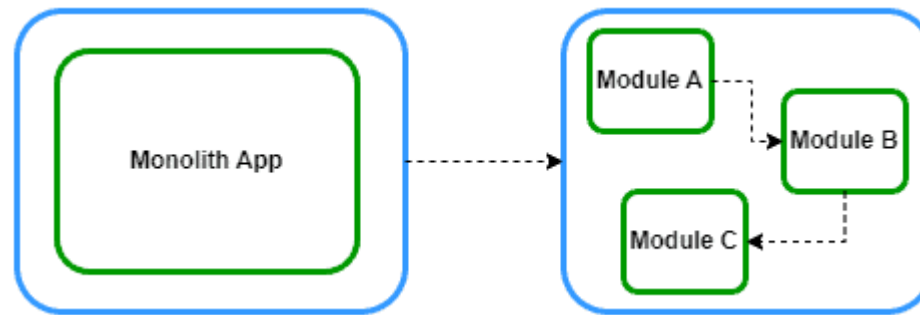
Java 17
Spring Native
Micrometer
Nouveaux starters
Modules Spring6



Application modulaire

Modular Monolith est un style architectural où notre code source est structuré sur le concept de modules.

Cela permet d'éventuellement migrer vers des architectures micro-services





Spring Modulith

Un module d'application est une unité de fonctionnalité qui expose une API à d'autres modules.

- Regroupé dans un sous-packages direct du package principal de l'application.
- Un module peut accéder au contenu de n'importe quel autre module, mais ne peut pas accéder aux sous-packages d'autres modules.

Spring Modulith permet

- la vérification
- La documentation (UML)
- Prise en compte des appels directs à l'API ou via bus de messages



Data persistence

Alternatives

JPA 3.0, Hibernate 6.1

Validation API

Caches



SQL vs NoSQL

SQL :

- Transactions ACID
- Intégrité des données
- SQL
- JDBC
- Support des outils et maturité

NoSQL

- Scalabilité, sharding, tolérance aux pannes
- Flexibilité
- Meilleur support du réactif
- Big Data
- Support spécialisés. Ex : Full-text



Spring et SQL

JDBC

- Requêtes SQLs complexes
- Réactif avec R2DBC

JPA/Hibernate

- ORM
- JPQL et API Criteria
- Lazy-loading
- Cache de second-niveau

Spring Data

- Génération des beans DAO
- Requêtes dynamiques sur convention de nommage
- Prise en charge automatique des transactions
- Support pour le test



Synthèse

Critère	JDBC	JPA	Spring Data JPA
Complexité	Haute	Moyenne	Basse
Code boilerplate	Élevé	Moyen	Faible
Contrôle	Total	Modéré	Faible
Performances	Optimales	Bonne	Bonne
Courbe d'apprentissage	Faible	Moyenne	Faible
Support des relations	Manuel	Automatique	Automatique
Maintenance	Complexe	Modérée	Facile
Requêtes complexes	Facile	Possible, mais moins naturel	Facile avec SQL natif ou <code>@Query</code>



Data persistence

Alternatives

JPA 3.0, Hibernate 6.1

Validation API

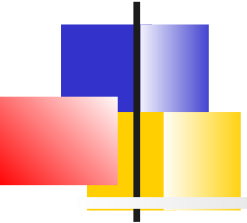
Caches



Versions

SB 3.0 : JPA 3.1, Hibernate 6.2.x

SB 3.1 : JPA 3.1 Hibernate 6.3.x



JPA 3.1

Classe UUID traitée comme un type Java de base

Extensions à JPQL et l'api Criteria

- Fonctions numériques :
Arrondis, exponentiel, logarithme, puissance
- Fonctions pour les LocalDate et Time
- Fonction extract pour extraire un entier d'une date



UUID

@Entity

```
public class Item {
```

```
    @ID @GeneratedValue(strategy=GenerationType.UUID)
```

```
    private java.util.UUID id;
```

```
    private String description;
```

```
    ...
```

```
}
```



Fonctions numériques

```
SELECT p.name as name,  
       CEILING(p.salary) as ceiling,  
       FLOOR(p.salary) as floor,  
       ROUND(p.salary, 1) as round,  
       EXP(p.yearsWorked) as exp,  
       LN(p.yearsWorked) as ln,  
       POWER(p.yearsWorked,2) as power,  
       SIGN(p.yearsWorked) as sign  
FROM Person p  
WHERE p.id=:id
```



LocalDate

```
SELECT p.name as name,  
       LOCAL TIME as localTime,  
       LOCAL DATETIME as localDateTime,  
       LOCAL DATE as localDate  
FROM Person p
```

Les champs *LocalDate*, *LocalTime* ou *LocalDateTime* sont mappés sans annotations particulières



EXTRACT

```
SELECT p.name as name,  
       EXTRACT(YEAR FROM p.birthDate) as year,  
       EXTRACT(QUARTER FROM p.birthDate) as quarter,  
       EXTRACT(MONTH FROM p.birthDate) as month,  
       EXTRACT(WEEK FROM p.birthDate) as week,  
       EXTRACT(DAY FROM p.birthDate) as day,  
       EXTRACT(HOUR FROM p.birthDate) as hour,  
       EXTRACT(MINUTE FROM p.birthDate) as minute,  
       EXTRACT(SECOND FROM p.birthDate) as second  
FROM Person p
```



Hibernate 6.3

Simplification des requêtes HQL/JPQL

Prise en charge des nouveaux types des BD

Ex Postgres : Support natif pour des types complexes, comme JSON et JSONB, et des opérateurs spécifiques comme ->> .

Optimisation des entités immuable

Nommage des séquences

Générateur d'UUID explicite

Support pour le multi-tenancy

Amélioration des performance



Requêtes HQL/JPQL

Avant

```
Query query = session.createQuery("FROM User WHERE name = :name");  
query.setParameter("name", "Alice");  
List<User> users = query.list();
```

Maintenant :

```
Query<User> query =  
    session.createQuery("FROM User WHERE name = :name", User.class);  
query.setParameter("name", "Alice");  
List<User> users = query.list();
```



Exemple JSON, Long Text

```
@Entity
@Table(name = "visits")
public class Visit extends BaseEntity {

    @Column(name = "recipe")
    @JdbcTypeCode(SqlTypes.JSON)
    private Recipe recipe;
}

----- Long text
@JdbcTypeCode(SqlTypes.LONGVARCHAR)
@Column(name = "description")
private String description;

@Column(name = "doc_txt", length = Length.LOB_DEFAULT)
private String docText;
```



@Immutable

Il est possible d'annoter une entité ou une relation avec **@Immutable**

=> Mise à jour non effectués

=> Optimisation

```
@Entity
@Immutable
@Table(name = "events_generated")
public class EventGeneratedId {
    ...
    @Immutable
    public Set<String> getGuestList() {
        return guestList;
    }
}
```



Séquence

```
// Avec hibernate 5 la séquence s'appelle hibernate_sequence
// Avec hibernate 6, visits_seq
@Entity
@Table(name = "visits")
public class Visit {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Integer id;
}
```

Hibernate 6 définit 3 stratégies de nommage :

- SingleNamingStrategy (Hibernate version < 5.3)
- LegacyNamingStrategy
- StandardNamingStrategy (Hibernate version >= 6.0)



Générateur UUID

La méthode recommandée est de définir l'algorithme de génération de l'UUID

```
@Id  
@UuidGenerator(style = UuidGenerator.Style.TIME)  
@GeneratedValue  
private UUID id;
```



Multi-tenancy

```
@Entity
@Table(name = "owner")
public class Owner {
    ...
    @TenantId
    @Column(name = "clinic")
    private String clinic;
}
```

Hibernate ajoute alors la condition where pour récupérer les données d'une clinic particulière.
Il faut fournir un resolver fournissant une valeur pour le tenant.

```
@Component
public class TenantIdentifierResolver implements CurrentTenantIdentifierResolver,
    HibernatePropertiesCustomizer {

    @Override
    public String resolveCurrentTenantIdentifier() {
        return ...//Fetch identifier from the session, request, etc.;
    }

    @Override
    public void customize(Map<String, Object> hibernateProperties) {
        hibernateProperties.put(AvailableSettings.MULTI_TENANT_IDENTIFIER_RESOLVER, this);
    }
}
```




Data persistence

Alternatives
JPA 3.0, Hibernate 6.1
Validation API
Caches



jakarta

Spring Boot 3 migrant vers Jakarta EE, l'API
Validation passe de javax à jakarta

Fonctionnalités identiques mais

- support du natif
- Support du réactif
- *@Validated* sur classe Service
- Personnalisation des messages d'erreur dans *ValidationMessages.properties*
- Meilleure intégration avec Jackson
- Support des Record



Data persistence

Alternatives
JPA 3.0, Hibernate 6.1
Validation API
Caches



Starter cache

Le starter cache permet d'ajouter de façon transparente des fonctionnalités de cache

La dépendance + l'annotation

@EnableCaching permet l'auto configuration d'un gestionnaire de cache

Une implémentation peut être fournie :

EhCache3, Hazelcast, InfiniSpan,
CouchBase, Redis, Caffeine, Cache2K,
ConcurrentHashMap



Configuration

La configuration du gestionnaire cache dépend de l'implémentation.

Exemple *ehcache.xml* :

```
<config xmlns="http://www.ehcache.org/v3">
  <cache alias="entitiesCache">
    <key-type>java.lang.String</key-type>
    <value-type>java.lang.Object</value-type>
    <expiry>
      <ttl unit="seconds">3600</ttl>
    </expiry>
    <heap unit="entries">1000</heap>
  </cache>
</config>
```



Annotations

@Cacheable(value="cacheName", key="entryKey")

- Sur une classe Repository ou Service. La méthode n'est pas appelée si la clé est présente dans le cache
- Au niveau d'une classe Entité, l'entité n'est pas chargée si elle est présente dans le cache



Cache de méthode

```
@Service
public class UserService {

    @Cacheable(value = "users", key = "#id")
    public User getUserById(Long id) {
        return userRepository.findById(id).orElseThrow(() -> new
RuntimeException("User not found"));
    }
}
```



Cache d'entité

Configuration :

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=
    org.hibernate.cache.jcache.JCacheRegionFactory
spring.jpa.properties.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider
```

Annotation sur l'entité :

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class User {
```

CacheConcurrencyStrategy : définit la stratégie de gestion des données dans le cache.

- **READ_ONLY** : pour des données immuables.
- **READ_WRITE** : pour des données fréquemment modifiées.
- **NONSTRICT_READ_WRITE** : pour des données rarement modifiées.



Mise à jour

L'annotation **@CachePut** permet d'exécuter une méthode dont le résultat est mis dans le cache.

Typiquement, à positionner sur une mise à jour d'entité

```
@CachePut(value = "users", key = "#user.id")  
public User updateUser(User user) {  
    return userRepository.save(user);  
}
```



Eviction

L'annotation **@CacheEvict** permet d'enlever une entrée ou toutes les entrées du cache

- Utilisé lors d'une suppression
- Ou pour libérer de la place dans le cache

```
@CacheEvict(value = "users", key = "#id")
public void deleteUserById(Long id) {
    userRepository.deleteById(id);
}
...
@CacheEvict(value = "users", allEntries = true)
public void clearCache() {
    // Nettoie tout le cache "users"
}
```



Monitoring

Actuator permet d'accéder aux données de configuration des cache et de nettoyer les caches

GET /actuator/caches

GET /actuator/caches/{name}

DELETE /actuator/caches

DELETE /actuator/caches/{name}



Statistiques

Si le cache est configuré pour collecter des statistiques, ils peuvent être remontés dans Actuator.

Config *ehcache.xml*

```
<cache alias="users"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToLiveSeconds="300"
      statistics="true">
</cache>
```



Remontée dans actuator

```
management.endpoints.web.exposure.include=health,metrics  
management.metrics.enable.cache=true
```

GET /actuator/metrics/cache.users



Couche web

Impératif / Reactif / Virtual Thread

Spring WebFlux

RestClient / WebClient

SSE, WebSocket



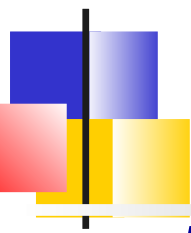
Reactive stack : Non blocking IO

La programmation réactive s'appuie sur les entrées/sorties non bloquantes.

L'idée est simple :

Récupérer des ressources qui seraient en attente d'E/S.

=> Un client d'un flux de données est informé lors les nouvelles données sont prêtes au lieu de les attendre



Eco-système Reactive Spring

Reactor est la brique de base de la pile réactive de Spring.

Il est utilisé dans tous les projets réactifs :

- **Spring WebFlux** des services web back-end réactifs. Alternative à *Spring MVC*
- **Spring Data** : *MongoDB, Cassandra, R2DBC*
- **Spring Reactive Security** : Sécurité réactive
- **Spring Test** : Support pour le réactif
- **Spring Cloud** : Projets micro-services. En particulier *Spring Cloud Data Stream, Spring Gateway, ...*



2 Types

Reactor offre principalement 2 types Java :

- ***Mono*** : Flux de 0..1 éléments
- ***Flux*** : Flux de 0..N éléments

Tous les 2 sont des implémentations de
l'interface ***Publisher*** de *Reactive Stream*

Mono offre un sous-ensemble des
opérateurs de Flux



Méthodes des abonnés

Un ***Flux<T>*** représente une séquence asynchrone de 0 à N événements, optionnellement terminée par un signal de fin ou une erreur.

Mono<T> représente une séquence de 0 à 1 événement, optionnellement terminée par un signal de fin ou une erreur

Les événements sont traduits par des appels de méthode sur les abonnés :

- Nouvelle valeur : *onNext()*
- Signal de fin : *onComplete()*
- Erreur : *onError()*



Pile réactive

L'utilisation d'une API non bloquante doit se faire dans toutes les couches de l'application :

- Persistance
- Service
- Web



Exemple ReactiveMongoRepository

```
import
    org.springframework.data.repository.reactive.ReactiveCrudRepository;
import reactor.core.publisher.Flux;

public interface PersonRepository extends ReactiveCrudRepository<Person,
    String> {
    // Requête personnalisée pour trouver les personnes par âge
    Flux<Person> findByAgeGreaterThan(int age);
}
```



Bean Service

```
@Service
public class PersonService {

    private final PersonRepository personRepository;

    public PersonService(PersonRepository personRepository) { this.personRepository = personRepository; }

    public Flux<Person> getAllPersons() { return personRepository.findAll(); }

    public Mono<Person> getPersonById(String id) { return personRepository.findById(id); }

    public Mono<Person> createPerson(Person person) { return personRepository.save(person); }

    public Mono<Person> updatePerson(String id, Person updatedPerson) {
        return personRepository.findById(id)
            .flatMap(existingPerson -> {
                existingPerson.setName(updatedPerson.getName());
                existingPerson.setAge(updatedPerson.getAge());
                return personRepository.save(existingPerson);
            });
    }

    public Mono<Void> deletePerson(String id) { return personRepository.deleteById(id); }

    public Flux<Person> getPersonsByAgeGreaterThan(int age) {
        return personRepository.findByAgeGreaterThan(age); }
}
```



Contrôleur

```
@RestController
@RequestMapping("/persons")
public class PersonController {

    private final PersonService personService;

    public PersonController(PersonService personService) { this.personService = personService; }

    @GetMapping
    public Flux<Person> getAllPersons() { return personService.getAllPersons(); }

    @GetMapping("/{id}")
    public Mono<Person> getPersonById(@PathVariable String id) { return personService.getPersonById(id); }

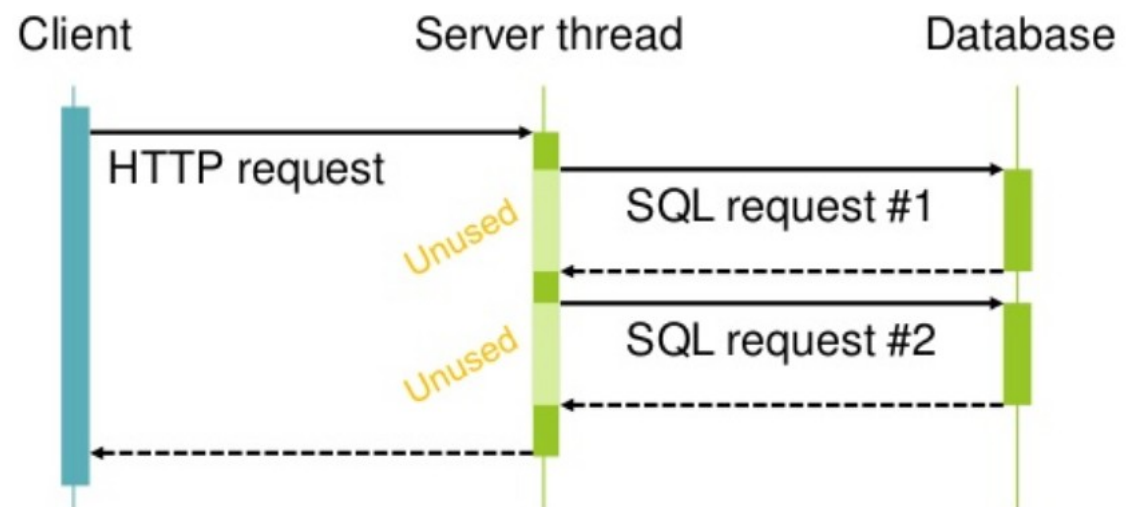
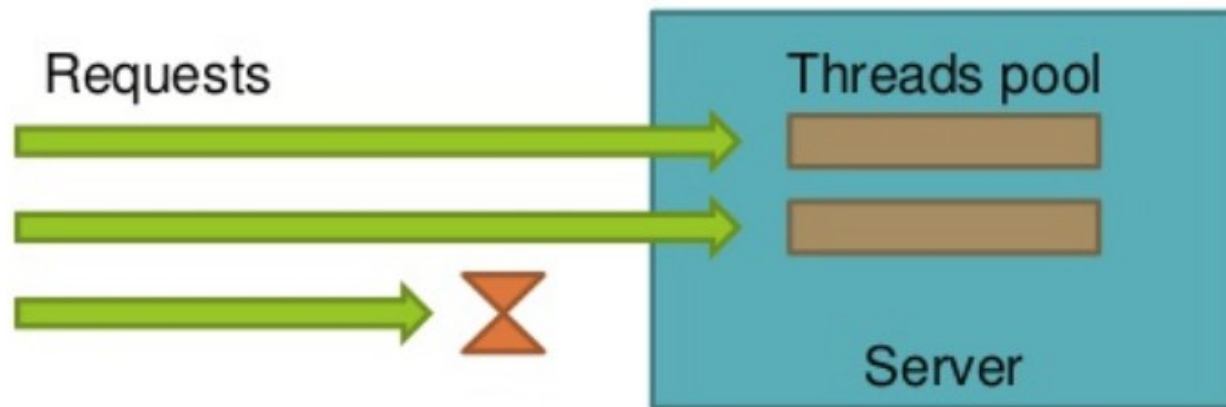
    @PostMapping
    public Mono<Person> createPerson(@RequestBody Person person) { return personService.createPerson(person); }

    @PutMapping("/{id}")
    public Mono<Person> updatePerson(@PathVariable String id, @RequestBody Person person) {
        return personService.updatePerson(id, person); }

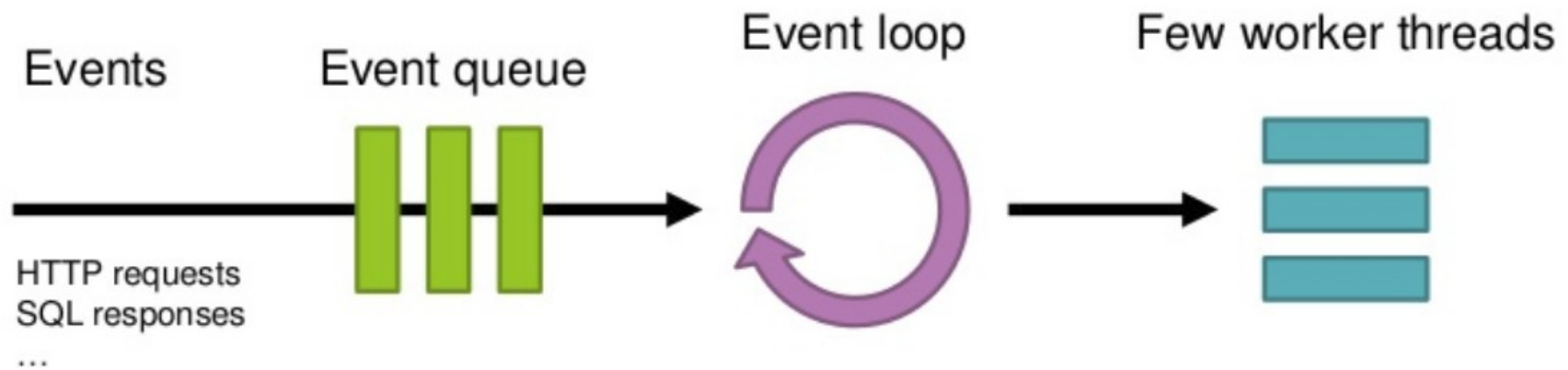
    @DeleteMapping("/{id}")
    public Mono<Void> deletePerson(@PathVariable String id) { return personService.deletePerson(id); }

    @GetMapping("/age/{age}")
    public Flux<Person> getPersonsByAgeGreaterThan(@PathVariable int age) {
        return personService.getPersonsByAgeGreaterThan(age); }
}
```

Modèle bloquant



Modèle non bloquant





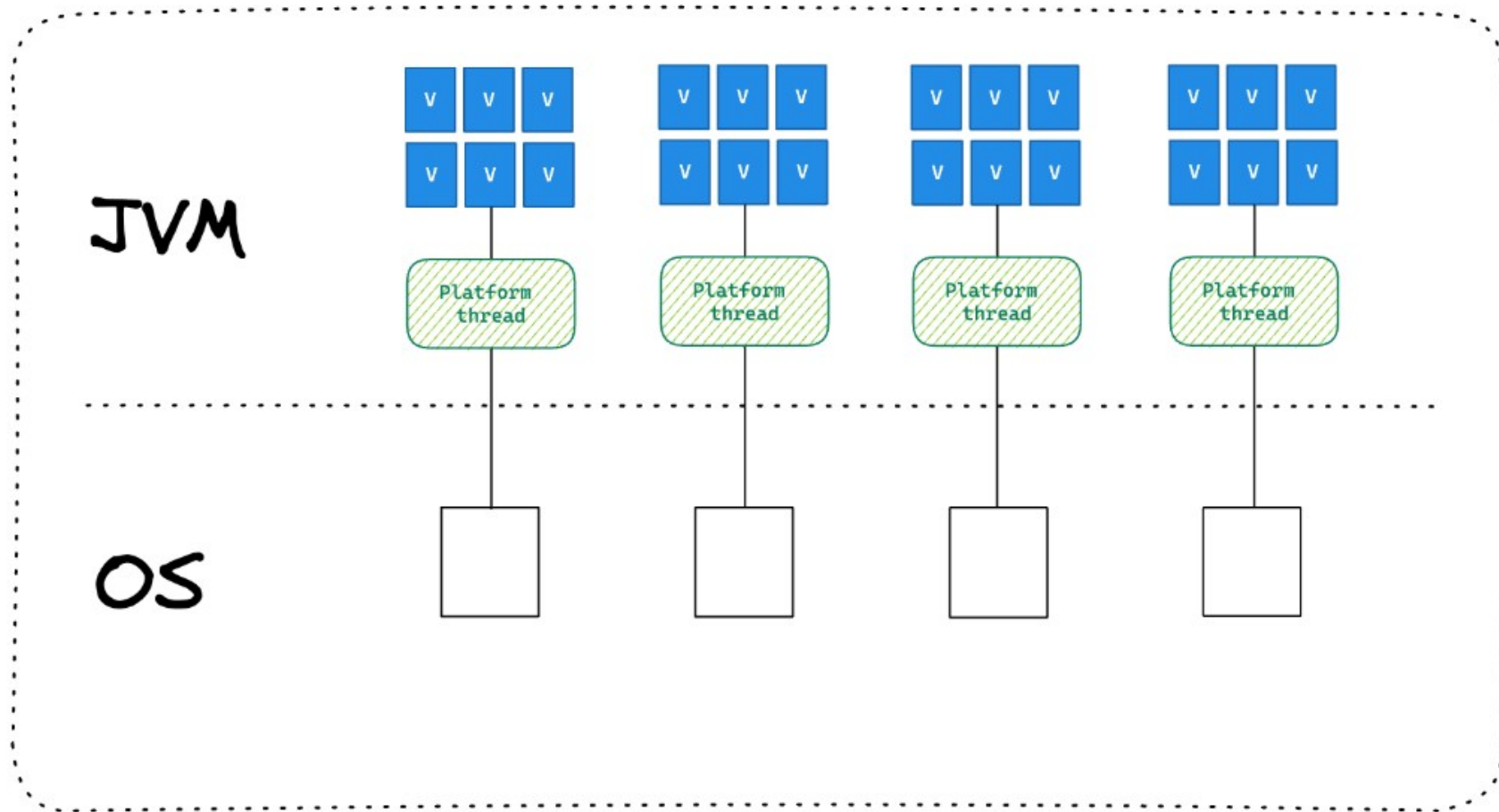
VirtualThread

Un **Virtual Thread** est une implémentation légère de thread en Java.

Contrairement aux threads classiques (aussi appelés Platform Threads), qui sont directement mappés sur des threads natifs du système d'exploitation, les Virtual Threads sont gérés entièrement par la JVM

- Lorsqu'un thread virtuel doit effectuer une opération bloquante, il est démonté du thread de la plateforme et déplacé vers la mémoire.
- Une fois l'opération de blocage terminée, il est retiré du tas et déplacé vers une liste d'attente pour le thread de la plateforme sur lequel il a été initialement monté.

Plus de limitation par le hardware





Utilisation dans SpringBoot

Si l'application nécessite de faire de nombreux appels I/O, il peut profiter des VirtualThreads.

Dans un contexte Spring MVC, il suffit de positionner :

spring.threads.virtual.enabled=true



Comparaison modèle réactif

Aspect	Spring WebFlux	Spring MVC avec Virtual Threads
Modèle	Réactif (non bloquant)	Impératif (bloquant) avec threads légers
Complexité	Complexe (requiert un paradigme réactif)	Simplicité du code impératif
Performances	Excellent pour les I/O intensives	Bonne gestion des I/O avec simplicité
Support Virtual Threads	Utilise un thread pool optimisé (par défaut)	Peut bénéficier directement des Virtual Threads



Couche web

Impératif vs Reactif

API REST

Spring WebFlux

WebClient

SSE, WebSocket



Introduction

Starter ***spring-reactive-web***

Spring Webflux offre en plus 2 modèles de programmation :

- **Contrôleurs annotés** : Idem à Spring MVC avec les mêmes annotations.
Les méthodes des contrôleurs peuvent retourner des types réactifs, des arguments réactifs sont associés à *@RequestBody*.
- **Endpoints fonctionnels** : Programmation fonctionnelle basée sur les lambdas.
Idéal pour de petites librairies permettant de router et traiter des requêtes.
Dans ce cas, l'application est en charge du traitement de la requête du début à la fin.

Dans la version récente de SpringBoot on peut mixer du code réactif et du code impératif.



Serveurs

Spring WebFlux est supporté sur

- Tomcat, Jetty, et les conteneurs de Servlet 3.1+,
- Ainsi que les environnements non-Servlet comme *Netty* ou *Undertow*

Le même modèle de programmation est supporté sur tous ces serveurs

Avec *SpringBoot*, la configuration par défaut démarre un serveur embarqué Netty



API

Le package *org.springframework.web.server* fournit une API pour traiter les requêtes HTTP.

La requête est alors traitée par :

- Une chaîne de **WebExceptionHandler**
- Une chaîne de **WebFilter**
- Et un **WebHandler** (*DispatcherHandler*)

Les chaînes de filtres et de gestionnaires d'exception sont configurées par **WebFluxConfigurer**

DispatcherHandler délègue le traitement de la requête via les annotations *@Controller* ou les endpoints fonctionnels

D'autre part, SpringWebFlux ajoute un id de requête dans les traces



Example

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }

    @PostMapping("/person")
    Mono<Void> create(@RequestBody Publisher<Person> personStream) {
        return this.repository.save(personStream).then();
    }

    @GetMapping("/person")
    Flux<Person> list() {
        return this.repository.findAll();
    }

    @GetMapping("/person/{id}")
    Mono<Person> findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```



Filtre

Les filtres avec *SpringWebFlux* manipulent *ServerWebExchange* et la chaîne de filtre.

Exemple :

```
@Component
public class SecurityWebFilter implements WebFilter{
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain)
    {
        if(!exchange.getRequest().getQueryParams().containsKey("user")){
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
        }
        return chain.filter(exchange);
    }
}
```



Exceptions

Les classes *@Controller* et *@ControllerAdvice* peuvent avoir des méthodes annotés par ***@ExceptionHandler***

Ces méthodes sont responsable de générer la réponse en cas de déclenchement de l'exception.

```
@ExceptionHandler  
    public ResponseEntity<String> handle(IOException ex) {  
        // ...  
    }
```

SpringFlux fournit une implémentation : *WebFluxResponseStatusExceptionHandler* qui traite les exceptions de type *ResponseStatusException*.



Configuration

Dans la configuration Java, on peut implémenter un ***WebFluxConfigurer*** afin de surcharger la configuration par défaut.

Différentes méthodes peuvent alors être surchargées.

Citons :

addCorsMappings(CorsRegistry registry) qui permet de configurer globalement le CORS.



Exemple

@Configuration

@EnableWebFlux

public class WebConfig implements **WebFluxConfigurer** {

 @Override

 public void configureHttpMessageCodecs(ServerCodecConfigurer configurator) {
 // configure message conversion...

 }

 @Override

 public void addCorsMappings(CorsRegistry registry) {
 // configure CORS...

 }

 @Override

 public void configureViewResolvers(ViewResolverRegistry registry) {
 // configure view resolution for HTML rendering...

 }

}



Couche web

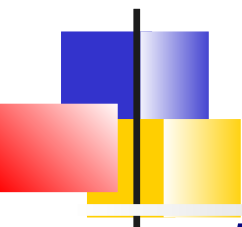
Impératif vs Reactif

API REST

Spring WebFlux

RestClient / WebClient

SSE, WebSocket



RestClient

RestClient est le successeur de *RestTemplate* et est dédié aux interactions synchrones

Spring fournit *RestClient.builder()* permettant de construire son client :

- baseUrl
- Cookies, headers
- Consumer
- Intercepteurs
- *ObservationRegistry*

Une fois construit, il permet d'effectuer des requêtes http et de récupérer :

- L'intégralité de la réponse *ResponseEntity*
- Seulement le corps (méthode *body*)



Exemple

```
import org.springframework.web.client.RestClient;

public class RestClientExample {
    private final RestClient restClient = RestClient.builder()
        .baseUrl("https://api.example.com")
        .build();

    public void callApi() {
        String response = restClient.get()
            .uri("/data")
            .retrieve()
            .body(String.class); // Appel bloquant
        System.out.println(response);
    }
}
```




Exception

Par défaut, RestClient génère une sous-classe de ***RestClientException*** lors de la récupération d'une réponse avec un code d'état 4xx ou 5xx.

Ce comportement peut être remplacé via ***onStatus***.

```
String result = restClient.get()
    .uri("https://example.com/this-url-does-not-exist")
    .retrieve()
    .onStatus(HttpStatusCode::is4xxClientError, (request, response) -> {
        throw new MyCustomRuntimeException(response.getStatusCode(),
        response.getHeaders());
    })
    .body(String.class);
```



Test

@RestClientTest Configure uniquement les beans nécessaires au test du RestClient et injecte un MockRestServiceServer pour simuler des réponses HTTP

MockRestServiceServer : Simule une API distante en contrôlant les réponses des requêtes.



Exemple

```
@RestClientTest(MyService.class)
```

```
class MyServiceTest {
```

```
    @Autowired
```

```
    private MyService myService;
```

```
    @Autowired
```

```
    private MockRestServiceServer mockServer;
```

```
    @Test
```

```
    void testGetSomething() {
```

```
        mockServer.expect(requestTo("http://example.org/resource"))
```

```
            .andRespond(withStatus(HttpStatus.OK).body("Hello World"));
```

```
        String response = myService.getSomething();
```

```
        assertEquals("Hello World", response);
```

```
        mockServer.verify();
```

```
    }
```

```
}
```



WebClient

WebFlux inclut ***WebClient*** facilitant les interactions REST¹

- Expose les I/O réseau via *ClientHttpRequest* et *ClientHttpResponse*.
 - Les corps de la requête et de la réponse sont des *Flux<DataBuffer>* plutôt que des *InputStream* et *OutputStream*.
- Même mécanismes de sérialisation (JSON, XML) permettant de travailler avec des objets typés.
- SB fournit un builder

1. Alternative non-bloquante à *RestClient*



Réponse

La méthode ***retrieve()*** permet de récupérer une réponse et de la décoder :

```
WebClient client = WebClient.builder()  
    .baseUrl("http://example.org")  
    .build();
```

```
Mono<Person> result = client.get()  
    .uri("/persons/{id}", id)  
    .accept(MediaType.APPLICATION_JSON)  
    .retrieve()  
    .bodyToMono(Person.class);
```



Exceptions

Les réponses avec des statuts 4xx ou 5xx provoquent une ***WebClientResponseException***.

Il est possible d'utiliser la méthode ***onStatus*** pour personnaliser le traitement de l'exception:

```
Mono<Person> result = client.get()
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .onStatus(HttpStatus::is4xxServerError, response -> ...)
    .onStatus(HttpStatus::is5xxServerError, response -> ...)
    .bodyToMono(Person.class);
```



Contrôle de la réponse

La méthode ***exchange()*** permet un meilleur contrôle de la réponse en permettant d'avoir un accès à *ClientResponse*

```
Mono<Person> result = client.get()  
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)  
    .exchange()  
    .flatMap(response -> response.bodyToMono(Person.class));
```



Corps de requête

Le corps de la requête peut être encodé à partir d'un *Mono*, d'un *Flux* ou d'une type simple:

```
Mono<Person> personMono = ... ;
```

```
Mono<Void> result = client.post()  
    .uri("/persons/{id}", id)  
    .contentType(MediaType.APPLICATION_JSON)  
    .body(personMono, Person.class)  
    .retrieve()  
    .bodyToMono(Void.class);
```




Corps de la requête (2)

```
Flux<Person> personFlux = ... ;
```

```
Mono<Void> result = client.post()  
    .uri("/persons/{id}", id)  
    .contentType(MediaType.APPLICATION_STREAM_JSON)  
    .body(personFlux, Person.class)  
    .retrieve()  
    .bodyToMono(Void.class);
```

```
Person person = ... ;
```

```
Mono<Void> result = client.post()  
    .uri("/persons/{id}", id)  
    .contentType(MediaType.APPLICATION_JSON)  
    .body(person, Person.class)  
    .retrieve()  
    .bodyToMono(Void.class);
```



Filtre client

Le ***WebClient.Builder*** permet
d'enregistrer un filtre qui intercepte les
requêtes

Exemple : Authentification Basique

```
WebClient client = WebClient.builder()  
    .filter(basicAuthentication("user", "password"))  
    .build();
```



Support pour Webflux

Spring5 propose des implémentations mock de *ServerHttpRequest* , *ServerHttpResponse*, et *ServerWebExchange* à utiliser dans des applications WebFlux.

Le ***WebTestClient*** permet de tester une application WebFlux

- Sans serveur : Test « unitaire »
- Avec serveur : Test e2e



Mise en place

La création d'un ***WebTestClient*** s'effectue de différentes façons.

En fonction du type de test que l'on veut effectuer on peut l'associer à :

- Un unique contrôleur
- Une *RouterFunction*
- Un contexte Spring
- Un serveur



Différents bind

Sans serveur HTTP avec des requêtes et des réponses mockées.

```
client = WebTestClient.bindToController(new TestController()).build();
```

```
RouterFunction<?> route = ...
```

```
    client = WebTestClient.bindToRouterFunction(route).build();
```

```
client = WebTestClient.bindToApplicationContext(context).build();
```

Connexion à un serveur démarré

```
client = WebTestClient.bindToServer().  
    baseUrl("http://localhost:8080").build();
```



Ecriture des tests

WebTestClient fournit une API identique à *WebClient*

L'exécution d'une requête peut se faire par ***exchange()***

Ensuite, une chaîne de vérification peut être combinée.

- Tester le statut et les entêtes
- Extraire le corps de la réponse et y effectuer des assertions



Example

```
client.get().uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON_UTF8)
    .exchange()
    .expectStatus().isOk()
    .expectHeader()
        .contentType(MediaType.APPLICATION_JSON_UTF8)
    .expectBodyList(Person.class) // Extraction
    .hasSize(3).contains(person); // Assertions
```



Réponses Stream

Le test des réponses sous forme de flux, s'effectue en 2 étapes :

- Récupérer un Flux via la méthode *returnResult*
- Utiliser StepVerifier de Reactor pour vérifier les événements du Flux.



Exemple

```
FluxExchangeResult<MyEvent> result = client.get().uri("/events")
    .accept(TEXT_EVENT_STREAM)
    .exchange()
    .expectStatus().isOk()
    .returnResult(MyEvent.class);
```

```
Flux<Event> eventFux = result.getResponseBody();
```

```
StepVerifier.create(eventFlux)
    .expectNext(person)
    .expectNextCount(4)
    .consumeNextWith(p -> ...)
    .thenCancel()
    .verify();
```



Couche web

Impératif vs Reactif

API REST

Spring WebFlux

WebClient

SSE, WebSocket



Server-side events

SSE permet une communication asynchrone d'un serveur vers un client via HTTP.

- Le serveur peut envoyer un flux d'événements qui met à jour le client de manière asynchrone.
- Presque tous les navigateurs supportent le SSE
- Il est très simple de s'abonner au flux en Javascript



Spring Webflux et SSE

Il est possible d'implémenter SSE dans une méthode contrôleur en renvoyant un *Flux* ou une entité *ServerSentEvent*

- Avec le Flux, il faut préciser le mime-type

`MediaType.TEXT_EVENT_STREAM_VALUE`

Spring offre également du support pour le WebClient



Examples

```
@GetMapping(path = "/stream-flux", produces =  
    MediaType.TEXT_EVENT_STREAM_VALUE)  
public Flux<String> streamFlux() {  
    return Flux.interval(Duration.ofSeconds(1))  
        .map(sequence -> "Flux - " + LocalDateTime.now().toString());  
}
```

```
@GetMapping("/stream-sse")  
public Flux<ServerSentEvent<String>> streamEvents() {  
    return Flux.interval(Duration.ofSeconds(1))  
        .map(sequence -> ServerSentEvent.<String> builder()  
            .id(String.valueOf(sequence))  
            .event("periodic-event")  
            .data("SSE - " + LocalDateTime.now().toString())  
            .build());  
}
```



Exemple *WebClient*

```
public void consumeServerSentEvent() {
    WebClient client = WebClient.create("http://localhost:8080/sse-server");
    ParameterizedTypeReference<ServerSentEvent<String>> type
    = new ParameterizedTypeReference<ServerSentEvent<String>>() {};

    Flux<ServerSentEvent<String>> eventStream = client.get()
        .uri("/stream-sse")
        .retrieve()
        .bodyToFlux(type);

    eventStream.subscribe(
        content -> logger.info("Time: {} - event: name[{}], id [{}], content[{}] ",
            LocalTime.now(), content.event(), content.id(), content.data()),
        error -> logger.error("Error receiving SSE: {}", error),
        () -> logger.info("Completed!!!"));
}
```



Exemple Javascript

<script>

```
var source = new EventSource("/orders/status");  
source.onmessage = (event) => {  
    console.log("An event "+event);
```

```
    var json = JSON.parse(event.data);  
    console.log("JSON "+JSON.stringify(json));
```

```
};  
</script>
```



Introduction

Le protocole **WebSocket** (RFC 6455) définit un standard pour une communication full-duplex entre un client et un serveur.

C'est un protocole TCP différent de HTTP mais qui utilise les ports 80 et 443 pour passer les firewall.

Spring Framework fournit une API WebSocket permettant d'écrire du code client ou serveur.



Cas d'usage

Une combinaison d'Ajax et de streaming HTTP ou du polling peuvent souvent avoir les mêmes effets

Les *WebSockets* sont utilisées lorsque le client et le serveur doivent échanger des événements à une haute fréquence avec peu de latence.

Attention, la configuration de beaucoup de proxy bloque les websockets !!



URL unique

A la différence des architectures HTTP ou REST, les *WebSockets* n'utilisent qu'**une seule URL** pour la connexion initiale

Tous les messages applicatifs utilisent alors la même connexion TCP.

=> Cela conduit à une architecture asynchrone et piloté par les événements



Mise en place

La mise en place consiste à :

- Définir côté serveur un ***WebSocketHandler***
- L'associer à une URL via un ***HandlerMapping*** et un ***WebSocketHandlerAdapter***
- Utiliser un ***WebSocketClient*** pour démarrer un session



WebSocketHandler

WebSocketHandler définit la méthode *handle()* qui prend une *WebSocketSession* et retourne *Mono<Void>* lorsque le traitement de la session est terminée

La session est traitée via 2 streams de *WebSocketMessage* :
1 pour le message d'entrée, 1 pour le message de sortie

La session propose donc 2 méthodes :

- ***Flux<WebSocketMessage> receive()*** : Accès au flux d'entrée se termine à la fermeture de connexion.
- ***Mono<Void> send(Publisher<WebSocketMessage>)*** : Prend un source pour les messages de sortie, écrit les messages et retourne *Mono<Void>* lorsque la source est tarie.



Exemple

```
@Component
public class ReactiveWebSocketHandler implements WebSocketHandler {

    // private fields ...

    @Override
    public Mono<Void> handle(WebSocketSession webSocketSession) {
        return webSocketSession.send(intervalFlux
            .map(webSocketSession::textMessage))
            .and(webSocketSession.receive()
                .map(WebSocketMessage::getPayloadAsText)
                .log());
    }
}
```



Mapping

@Autowired

```
private WebSocketHandler webSocketHandler;
```

@Bean

```
public HandlerMapping webSocketHandlerMapping() {  
    Map<String, WebSocketHandler> map = new HashMap<>();  
    map.put("/event-emitter", webSocketHandler);  
  
    SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();  
    handlerMapping.setOrder(1);  
    handlerMapping.setUrlMap(map);  
    return handlerMapping;  
}
```

@Bean

```
public WebSocketHandlerAdapter handlerAdapter() {  
    return new WebSocketHandlerAdapter();  
}
```



Client

Spring WebFlux fournit une interface ***WebSocketClient*** avec des implémentations pour Reactor Netty, Tomcat, Jetty, Undertow, et Java standard (i.e. JSR-356).

Pour démarrer une session *WebSocket*, créer une instance du client et utiliser sa méthode ***execute*** :

```
WebSocketClient client = new ReactorNettyWebSocketClient();
```

```
URI url = new URI("ws://localhost:8080/path");  
client.execute(url, session ->  
    session.receive()  
        .doOnNext(System.out::println)  
        .then());
```



Exemple avec envoi de message vers le serveur

```
WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute(
    URI.create("ws://localhost:8080/event-emitter"),
    session -> {
        // Receive messages and for each message, send a response
        Flux<Void> receiveAndRespond = session.receive()
            .map(webSocketMessage -> webSocketMessage.getPayloadAsText())
            .flatMap(message -> {
                System.out.println("Received: " + message);
                return session.send(Mono.just(session.textMessage("response
to: " + message))).then());
            })
            .log();
        return receiveAndRespond.then();
    })
    .block(Duration.ofSeconds(60));
```




Sécurité

What's new

Modèle Réactif

Support OpenID/oauth2



Nouveautés

Migration vers Jakarta EE 10

Suppression config XML et *NoOpPasswordEncoder*

Configuration via des beans explicites

AuthorizeHttpRequest plutôt que *authorizeRequest*

RequestMatcher remplace *AntMatcher*, *MvcMatcher*,
RegexMatcher

Support des Virtual Thread

Support des en-têtes de sécurité avancés *Cross-Origin-Embedder-Policy* et *Cross-Origin-Opener-Policy*.

Gestion améliorée des CORS



SecurityFilterChain

Au lieu d'utiliser `WebSecurityConfigurerAdapter` et sa méthode configure la chaîne de filtre est construite dans une méthode `@Bean` en utilisant `HttpSecurity`

```
@Configuration
public class SecurityConfiguration {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authz) -> authz
                .anyRequest().authenticated()
            )
            .httpBasic(withDefaults());
        return http.build();
    }
}
```



Ignorer la sécurité

Au lieu d'utiliser *WebSecurityConfigurerAdapter*, les chemins à ignorer sont configuré via un bean *WebSecurityCustomizer*

```
@Configuration
public class SecurityConfiguration {

    @Bean
    public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) ->
            web.ignoring().antMatchers("/ignore1", "/ignore2");
    }
}
```



Méthodes de httpsecurity

Les méthodes de *HttpSecurity* permettant de configurer le filtre prennent en argument un lambda.

La configuration par défaut peut être indiquée via :

- *Customizer.withDefaults()*



RequestMatcher

Les classes AntMatcher, MvcMatcher, and RegexMatcher sont dépréciées.

```
@EnableWebSecurity
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        return http.authorizeHttpRequests(auth -> auth
            .requestMatchers("/greet").permitAll()
            .anyRequest().authenticated())
            .formLogin()
            .build();
    }
}
```



Gestion du CORS

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .cors(cors -> cors.configurationSource(corsConfigurationSource()))
        .authorizeHttpRequests(authorize -> authorize.anyRequest().authenticated());
    return http.build();
}
```

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.addAllowedOrigin("https://example.com");
    configuration.addAllowedMethod("*");
    configuration.addAllowedHeader("*");
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```



@EnableMethodSecurity

@EnableMethodSecurity améliore

@EnableGlobalMethodSecurity de plusieurs façons:

- Utilise l'API AuthorizationManager simplifiée .
- Favorise la configuration directe basée sur les beans, au lieu de nécessiter l'extension de GlobalMethodSecurityConfiguration pour personnaliser les beans
- Est construit à l'aide de Spring AOP natif, supprimant les abstractions et vous permettant d'utiliser les blocs de construction Spring AOP pour personnaliser
- Vérifie les annotations en conflit pour garantir une configuration de sécurité sans ambiguïté
- Conforme à JSR-250
- Active *@PreAuthorize*, *@PostAuthorize*, *@PreFilter* et *@PostFilter* par défaut



@EnableMethodSecurity

Par défaut *@EnableMethodSecurity* permet d'annoter les méthodes de la couche service via :

@PreAuthorize, ***@PostAuthorize***, ***@PreFilter***, et
@PostFilter

@EnableMethodSecurity(securedEnabled=true) permet d'utiliser l'annotation Spring :

@Secured

@EnableMethodSecurity(jsr250Enabled = true) permet d'utiliser les annotations JavaEE :

@RolesAllowed, ***@PermitAll***, ***@DenyAll***



Annotations

Ces annotations utilisent des expressions SpEL pour vérifier les droits d'accès

- @PreAuthorize(Expr) : Vérifie l'expression avant d'entrer dans la méthode
- @PostAuthorize(Expr) : Vérifie l'expression au retour de la méthode qui peut utiliser la valeur retournée

```
@PostAuthorize("returnObject.username == authentication.principal.nickName")
```

```
public CustomUser loadUserDetail(String username) {
```

- @PreFilter(Expr) : filtre les collections en entrée d'une méthode

```
@PreFilter (value = "filterObject != authentication.principal.username", filterTarget = "usernames")
```

```
public String joinUsernamesAndRoles(String [] usernames, ...)
```

- @PostFilter(Expr) : filtre les collections en sortie d'une méthode

```
@PostFilter("filterObject != authentication.principal.username")
```

```
public List<String> getAllUsernamesExceptCurrent() {
```



Sécurité

What's new
Modèle Réactif
Support OpenID/oauth2



Introduction

Spring Security apporte les nouveautés suivantes pour la stack reactive:

- *@EnableWebFluxSecurity* : Configuration par défaut de la sécurité
- *@EnableReactiveMethodSecurity* :
Activation sécurité au niveau méthode
- *ReactiveUserDetailsService* : Intégration à un realm custom
- Test de WebFlux



WebFlux Security

La sécurité pour *Webflux* est consistante avec celle de Spring MVC

- Elle est implémentée sous forme de **filtre** (WebFilter) que l'on peut configurer finement
- L'annotation **@EnableWebFluxSecurity** permet d'avoir une configuration par défaut.
- La configuration par défaut peut être personnalisée via la classe **ServerHttpSecurity**

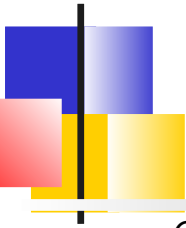
Les Beans de personnalisation sont réactifs



Configuration minimale

```
/* Configuration fournissant un authentification basique
 * via une page de login et de logout
 * Toutes les URLs sont protégées
 * Les entêtes HTTP relatifs à la sécurité sont positionnés (CSRF, ...)
 */
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {

    /*
     * MapReactiveUserDetailsService implémente ReactiveUserDetailsService
     */
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```



Configuration personnalisée

```
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {

    @Bean
    public SecurityWebFilterChain
        springSecurityFilterChain(ServerHttpSecurity http) {
        http.csrf(csrfSpec -> csrfSpec.disable())
            .authorizeExchange( acl -> acl
                .pathMatchers(HttpMethod.POST, "/accounts").hasRole("ADMIN")
                .pathMatchers("/accounts").authenticated()
                .anyExchange().permitAll())
            .formLogin(Customizer.withDefaults())

        return http.build();
    }
}
```



Sécurité au niveau des méthodes

En programmation réactive, la sécurité au niveau méthodes est possible en utilisant des **Reactor's context**

Elle peut être cumulée avec la sécurité sur les URLs

De la même façon, on obtient une configuration par défaut en utilisant l'annotation

@EnableReactiveMethodSecurity



Exemple minimal

@EnableReactiveMethodSecurity

```
public class SecurityConfig {  
    @Bean  
    public MapReactiveUserDetailsService userDetailsService() {  
        User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();  
        UserDetails rob =  
            userBuilder.username("rob").password("rob").roles("USER").build();  
        UserDetails admin =  
            userBuilder.username("admin").password("admin").roles("USER", "ADMIN").build();  
        return new MapReactiveUserDetailsService(rob, admin);  
    }  
}
```

@Component

```
public class HelloWorldMessageService {  
    @PreAuthorize("hasRole('ADMIN')")  
    public Mono<String> findMessage() {  
        return Mono.just("Hello World!");  
    }  
}
```



Test de la sécurité

Spring Security 5+ offre un support pour tester la sécurité pour tout type de combinaison :
Impératif/Réactif et URL/méthodes

Annotations : *@WithMockUser*,
@WithAnonymousUser, *@WithUserDetails*,
MockMvc

Dans l'univers réactif : Utilisation de
StepVerifier provenant de *Reactor*
permettant d'exprimer les événements
attendus d'un *Publisher* lors d'un abonnement



Sécurité

What's new
Modèle Réactif
Support OpenID/oAuth2



Starters SpringBoot oAuth

OAuth2 Client : Intégration pour utiliser un login oAuth2 fournit par Google, Github, Facebook, Keycloak ...

OAuth2 Resource server : Application permettant de définir des ACLs par rapport aux scopes client et aux rôles contenu dans des jetons oAuth

OAuth2 Authorization server : Serveur délivrant les jetons



OpenID avec SpringBoot

Spring Boot facilite la configuration des providers classiques : Google, Github, Facebook, Keycloak ...

@Bean

```
public SecurityWebFilterChain
    securityWebFilterChain(ServerHttpSecurity http) {
    return http.authorizeExchange()
        .anyExchange().authenticated()
        .and().oauth2Login()
        .and().csrf().disable()
        .build();
}
```



Configuration

Google

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: google-client-id
            client-secret: google-client-secret
```

Keycloak

```
spring :
  security:
    oauth2:
      client:
        provider:
          keycloak:
            issuer-uri: http://keycloak/realms/<realm-name>/
        registration:
          spring-app:
            provider: keycloak
            client-id: spring-app
            client-secret: 57abb4f6-5130-4c73-9545-6d377dd947cf
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
            scope :openid
            pkce-enabled: true
```



Accès à l'utilisateur loggé

```
@GetMapping("/oidc-principal")
public OidcUser getOidcUserPrincipal(
    @AuthenticationPrincipal OidcUser principal) {
    return principal;
}

...

Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
if (authentication.getPrincipal() instanceof OidcUser) {
    OidcUser principal = ((OidcUser)
        authentication.getPrincipal());

    // ...
}
```



Serveur de ressource

La dépendance et l'issuer uri suffisent à la configuration :

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://idp.example.com/issuer
```




Scope vs GrantedAuthority

Spring ajoute des Authority à l'Authentication en fonction des des scopes présent dans le jeton

- Par défaut, les autorités correspondant aux scopes sont préfixées par "SCOPE_".

Cela peut être adapté via le bean
JwtAuthenticationConverter

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();
    grantedAuthoritiesConverter.setAuthoritiesClaimName("authorities");

    JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter);
    return jwtAuthenticationConverter;
}
```



Configuration du SecurityFilterChain

```
@Configuration
@EnableWebSecurity
public class MyCustomSecurityConfiguration {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/messages/**").hasAuthority("SCOPE_message:read")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    .jwtAuthenticationConverter(myConverter())
                )
            );
        return http.build();
    }
}
```



oauthServer

Le starter oauthserver permet de délivrer des jetons à des clients.

Les endpoints oids sont sécurisés. Une configuration par défaut est disponible

La configuration des clients/users peut s'effectuer programmatiquement via `application.yml`



Exemple : configuration

```
spring:
  security:
    oauth2:
      authorizationserver:
        client:
          articles-client:
            registration:
              client-id: articles-client
              client-secret: "{noop}secret"
              client-name: Articles Client
              client-authentication-methods:
                - client_secret_basic
            authorization-grant-types:
              - authorization_code
              - refresh_token
            redirect-uris:
              - http://127.0.0.1:8080/login/oauth2/code/articles-client-oidc
              - http://127.0.0.1:8080/authorized
            scopes:
              - openid
              - articles.read
```



Spring et les tests

Spring Test

Apports de Spring Boot
Tests auto-configurés
Spring Cloud Contract



Versions

Spring/SpringBoot/JUnit

SpringBoot 1, Spring 4, JUnit4

Dernière version Septembre 2018

SpringBoot 2, Spring 5, JUnit5

Première version ~2018

SpringBoot 3, Spring 6, JUnit5

Première version Fin 2022



Rappels *spring-test*

Spring Test apporte peu pour le test unitaire

- **Mocking** de l'environnement en particulier l'API servlet ou Reactive
- Package **d'utilitaires** : *org.springframework.test.util*

Et beaucoup pour les tests d'intégration (impliquant un *ApplicationContext* Spring) :

- **Cache** du conteneur Spring pour accélérer les tests
- **Injection** des données de test
- Gestion de la **transaction** (roll-back)
- Des classes **utilitaires**
- **Intégration JUnit4 et JUnit5**



Intégration JUnit

- Pour JUnit4 :

`@RunWith(SpringJUnit4ClassRunner.class)`

ou `@RunWith(SpringRunner.class)`

Permet de charger un contexte Spring, effectuer l'injection de dépendances, etc.

- Pour JUnit5 :

`@ExtendWith(SpringExtension.class)`

Permet aussi de charger un contexte Spring, effectuer l'injection de dépendances, etc.

Et en plus de l'injection de dépendance pour les méthodes de test, des conditions d'exécution en fonction de la configuration Spring, des annotations supplémentaires pour gérer les transactions



Exemple JUnit5

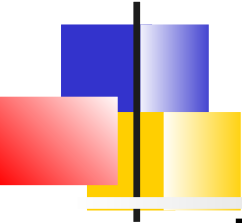
```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```



SpringBoot et les tests

Rappels Spring Test
Apports de Spring Boot
Tests auto-configurés
Spring Cloud Contract



spring-boot-starter-test

L'ajout de ***spring-boot-starter-test*** (dans le *scope test*), ajoute les dépendances suivantes :

- *Spring Test : Utilitaires Spring pour le Test*
- ***Spring Boot Test*** : *Utilitaire liant Spring Test à Spring Boot*
- ***Spring Boot Test Autoconfigure*** : *Tests auto-configurés*
- *JUnit4, AssertJ, Hamcrest* (SB 1.x) ou *JUnit5* (SB 2.X):
- *Mockito* : Un framework pour générer des classes Mock
- *JSONassert* : Une librairie pour les assertions JSON
- *JsonPath* : XPath pour JSON.



Annotations apportées

De nouvelles annotations sont disponibles via le starter :

- *@SpringBootTest* permettant de définir l'*ApplicationContext* Spring à utiliser pour un test grâce à un mécanisme de détection de configuration
- Annotations permettant des tests auto-configurés.
Ex : Auto-configuration pour tester des *RestController* en isolation
- Annotation permettant de créer des beans Mockito



@SpringBootTest

Il est possible d'utiliser l'annotation **@SpringBootTest** remplaçant la configuration standard de *spring-test* (*@ContextConfiguration*)

L'annotation crée le contexte applicatif (*ApplicationContext*) utilisé lors des tests en utilisant *SpringApplication* (classe principale)



Équivalence

```
// Annotations SpringBootTest
```

```
@RunWith(SpringRunner.class)
```

```
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
```

```
public class SpringBootTestApplicationTests {
```

```
// Annotations classiques
```

```
@RunWith(SpringRunner.class)
```

```
@SpringApplicationConfiguration(classes =  
    SprintBootTestApplication.class)
```

```
@WebAppConfiguration
```

```
public class SpringBootTestApplicationTests
```



Attribut Class

L'annotation `@SpringBootTest` peut préciser les classes de configuration utilisé pour charger le contexte applicatif via l'attribut ***classes***

Exemple :

```
@SpringBootTest(classes = ForumApp.class)
```



Attribut *WebEnvironment*

L'attribut *WebEnvironment* permet de préciser le type de contexte applicatif que l'on désire :

- **MOCK** : Fournit un environnement de serveur Mocké (le conteneur de servlet n'est pas démarré) : *WebApplicationContext*
- **RANDOM_PORT** : Charge un *ServletWebServerApplicationContext*. Le conteneur est démarré sur un port aléatoire
- **DEFINED_PORT** : Charge un *ServletWebServerApplicationContext*. Le conteneur est démarré sur un port spécifié
- **NONE** : Pas d'environnement servlet. *ApplicationContext* simple



Détection de la configuration

Les annotations **@*Test** servent comme point de départ pour la recherche de configuration.

Dans le cas de *SpringBootTest*, si l'attribut *class* n'est pas renseigné, l'algorithme cherche la première classe annotée *@SpringBootApplication* ou *@SpringBootConfiguration* en **remontant de packages**

=> Il est donc recommandé d'utiliser la même hiérarchie de package que le code principal



Mocking des beans

L'annotation **@MockBean** définit un bean Mockito

Cela permet de remplacer ou de créer de nouveaux beans

L'annotation peut être utilisée :

- Sur les classes de test
- Sur les champs de la classe de test, dans ce cas le bean mockito est injecté

Les beans Mockito sont automatiquement réinitialisés après chaque test



Exemple *MockBean*

```
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

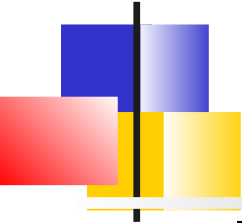
    @Autowired
    private Reverser reverser;

    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```



SpringBoot et les tests

Rappels Spring Test
Apports de Spring Boot
Tests auto-configurés
Spring Cloud Contract



Tests auto-configurés

Les capacités d'auto-configuration de Spring Boot peuvent ne pas être adaptées au test.

- Lorsque l'on teste la couche contrôleur, on n'a pas envie que SpringBoot nous démarre automatiquement une base de données

Le module *spring-boot-test-autoconfigure* incluent des annotations qui permettent de tester par couche les applications



Tests JSON

Afin de tester si la sérialisation JSON fonctionne correctement, l'annotation **@JsonTest** peut être utilisée.

Elle configure automatiquement l'environnement *Jackson* ou *Gson*

Les classes utilitaires *JacksonTester*, *GsonTester* ou *BasicJsonTester* peuvent être injectées et utilisées, les assertions spécifiques à JSON peuvent être utilisées



Example

@JsonTest

```
public class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    public void testSerialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a `.json` file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
        assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make")
            .isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
        assertThat(this.json.parse(content))
            .isEqualTo(new VehicleDetails("Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }
}
```



Tests de Spring MVC

L'annotation **@WebMvcTest** configure l'infrastructure Spring MVC et limite le scan aux annotations de Spring MVC

Elle configure également *MockMvc* qui permet de se passer d'un serveur Http complet

Pour les tests *Selenium* ou *HtmlUnit*, un client Web est également fourni



Example

```
@WebMvcTest(UserVehicleController.class)
```

```
public class MyControllerTests {
```

```
    @Autowired
```

```
    private MockMvc mvc;
```

```
    @MockBean
```

```
    private UserVehicleService userVehicleService;
```

```
    @Test
```

```
    public void testExample() throws Exception {
```

```
        given(this.userVehicleService.getVehicleDetails("sboot"))
```

```
            .willReturn(new VehicleDetails("Honda", "Civic"));
```

```
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
```

```
            .andExpect(status().isOk()).andExpect(content().string("Honda
```

```
Civic"));
```

```
    }
```

```
}
```



Example (2)

```
@WebMvcTest(UserVehicleController.class)
```

```
public class MyHtmlUnitTests {
```

```
    // WebClient is auto-configured thanks to HtmlUnit
```

```
    @Autowired
```

```
    private WebClient webClient;
```

```
    @MockBean
```

```
    private UserVehicleService userVehicleService;
```

```
    @Test
```

```
    public void testExample() throws Exception {
```

```
        given(this.userVehicleService.getVehicleDetails("sboot"))
```

```
            .willReturn(new VehicleDetails("Honda", "Civic"));
```

```
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
```

```
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
```

```
    }
```

```
}
```



Tests JPA

@DataJpaTest configure une base de donnée mémoire, scanne les *@Entity* et configure les Repository JPA

Les tests sont transactionnels et un rollback est effectué à la fin du test

- Possibilité de changer ce comportement par *@Transactional*

Un *TestEntityManager* peut être injecté ainsi qu'un *JdbcTemplate*



Example

@DataJpaTest

```
public class ExampleRepositoryTests {  
  
    @Autowired  
    private TestEntityManager entityManager;  
  
    @Autowired  
    private UserRepository repository;  
  
    @Test  
    public void testExample() throws Exception {  
        this.entityManager.persist(new User("sboot", "1234"));  
        User user = this.repository.findByUsername("sboot");  
        assertThat(user.getUsername()).isEqualTo("sboot");  
        assertThat(user.getVin()).isEqualTo("1234");  
    }  
}
```



Autres tests auto-configurés

@WebFluxTest : Test des contrôleurs Spring Webflux

@JdbcTest : Seulement la *datasource* et *jdbcTemplate*.

@JooqTest : Configure un *DSLContext*.

@DataMongoTest : Configure une base mémoire Mongo, *MongoTemplate*, scanne les classes *@Document* et configure les MongoDB repositories.

@DataRedisTest : Test des applications Redis applications.

@DataLdapTest : Serveur embarqué LDAP (if available), *LdapTemplate*, Classes *@Entry* et LDAP repositories

@RestClientTest : Test des clients REST. Jackson, GSON, ... + *RestTemplateBuilder*, et du support pour *MockRestServiceServer*.



Example

```
@RestClientTest(RestService.class)
public class RestserviceTest {
    @Autowired
    private MockRestServiceServer server;
    @Autowired
    private ObjectMapper objectMapper;
    @Autowired
    private RestService restService;

    @BeforeEach
    public void setUp() throws Exception {
        Member aMember = ...
        String memberString = objectMapper.writeValueAsString(aMember);

        this.server.expect(requestTo("/members/1"))
            .andRespond(withSuccess(memberString, MediaType.APPLICATION_JSON));
    }

    @Test
    public void whenCallingGetMember_thenOk() throws Exception {
        assertThat(restService.getMember(1)).extracting("email").isEqualTo("d@gmail.com");
    }
}
```



Test et sécurité

Spring propose plusieurs annotations pour exécuter les tests d'une application sécurisée par SpringSecurity.

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-test</artifactId>
<scope>test</scope>
</dependency>
```

@WithMockUser : Le test est exécuté avec un utilisateur dont on peut préciser les détails (login, password, rôles)

@WithAnonymousUser : Annote une méthode

@WithUserDetails("aLogin") : Le test est exécuté avec l'utilisateur chargé par *UserDetailsService*

@WithSecurityContext : Qui permet de créer le SecurityContext que l'on veut



SpringBoot et les tests

Rappels Spring Test
Apports de Spring Boot
Tests auto-configurés
Spring Cloud Contract



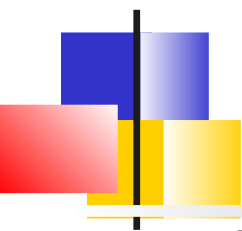
Consumer Driven Contract Test

Consumer-driven contract test Pattern¹ : Vérifier que la « forme » de l'API d'un service répond aux attentes du consommateur.

Dans le cas REST, le test de contrat vérifie que le fournisseur implémente un point de terminaison qui :

- A la méthode et le chemin HTTP attendus
- Accepte les entêtes attendus
- Accepte un corps de requête, le cas échéant
- Renvoie une réponse avec le code d'état, les entêtes et le corps attendus

1. <http://microservices.io/patterns/testing/service-integration-contract-test.html>



Spécification par l'exemple

Le *Consumer-driven Contract* définit les interactions entre un fournisseur et un consommateur via des **exemples**¹, i.e les contrats

Chaque contrat consiste en des exemples de messages échangés durant une interaction



Différents contrats

La structure d'un contrat dépend du type d'interaction entre les services

- *REST* : Des exemples de requêtes HTTP et les réponses attendues
- *Publish/subscribe* : Les événements du domaine
- *Requêtes /réponses asynchrones* :
Message de commande et de réponse



Spring Cloud Contract

Spring Cloud Contract est un projet qui permet d'adopter une approche *Consumer Driven Contract*

A partir d'une spécification d'interaction entre un producteur/serveur et consommateur/client, cela permet

- De générer des tests d'acceptation côté producteur
- De créer des mocks serveur pour les tests côté consommateur

Process

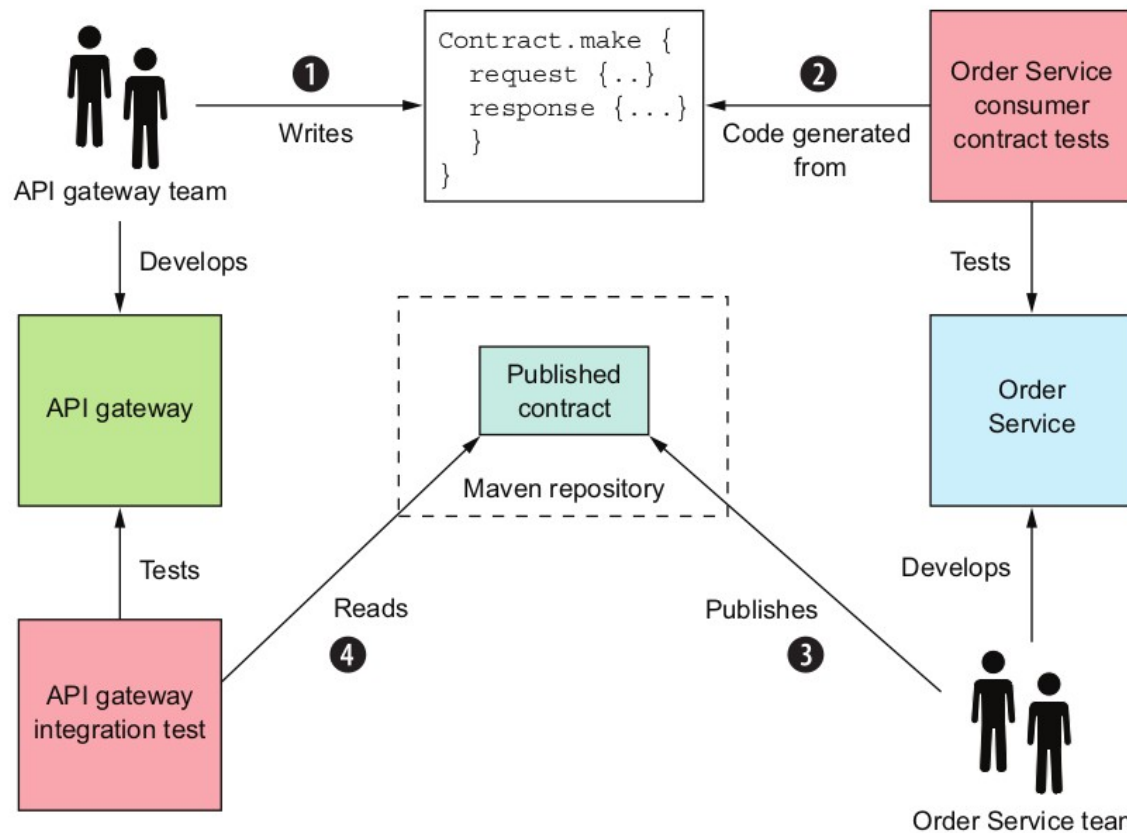


Figure 9.8 The API Gateway team writes the contracts. The Order Service team uses those contracts to test Order Service and publishes them to a repository. The API Gateway team uses the published contracts to test API Gateway.



Example Groovy

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    description "should return even when number input is even"
    request{
        method GET()
        url("/validate/prime-number") {
            queryParameters {
                parameter("number", "2")
            }
        }
    }
    response {
        body("Even")
        status 200
    }
}
```



Exemple : Génération des tests d'acceptation côté producteur

```
public class ContractVerifierTest extends BaseTestClass {

    @Test
    public void validate_shouldReturnEvenWhenRequestParamIsEven() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .queryParams("number", "2")
            .get("/validate/prime-number");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);

        // and:
        String responseBody = response.getBody().asString();
        assertThat(responseBody).isEqualTo("Even");
    }
}
```



Utilisation du mock serveur côté Consommateur

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
@AutoConfigureStubRunner(
    stubsMode = StubRunnerProperties.StubsMode.LOCAL,
    ids = "com.baeldung.spring.cloud:spring-cloud-contract-producer::stubs:8090")
public class BasicMathControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void given_WhenPassEvenNumberInQueryParam_ThenReturnEven()
        throws Exception {

        mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=2")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string("Even"));
    }
}
```




Interactions Publish/Subscribe

La spécification inclut le nom du canal de communication et la structure des messages.

Le contrat spécifie des exemples de message



Contrat

```
org.springframework.cloud.contract.spec.Contract.make {  
    label 'orderCreatedEvent'  
    input {  
        triggeredBy('orderCreated()')  
    }  
    outputMessage {  
        sentTo('order-channel')  
        body(''{"orderDetails":{"lineItems":[{"quantity":5,"menuItemId":"1",  
            "name":"Chicken Vindaloo",  
            "price":"12.34","total":"61.70"}]},  
            "orderTotal":"61.70",  
            "restaurantId":1,  
            "consumerId":1511300065921},"orderState":"APPROVAL_PENDING"}''')  
        headers {  
            header('event-aggregate-type','orderservice.domain.Order')  
            header('event-aggregate-id','1')  
        }  
    }  
}
```



Exécution des tests de composants

Pour exécuter les tests de composants, il faut disposer des infrastructures (BD, Message Broker) et de mock des services

Pour les infrastructures, les alternatives sont :

- Utiliser des BD/Message Broker in-memory
Attention on n'est plus dans les conditions de production
- Utiliser des images docker pour démarrer les services nécessaires
Plus lourd à mettre en place

Pour mocker les services dépendants, on peut utiliser les approches *Consumer Driven Contract* ou directement *WireMock*



Packaging et déploiement

Les plugins gradle et maven permettent plusieurs formats de packaging :

- Uber jar : Jar autoexécutable avec facilité pour mise en service
- Docker : Image docker construite avec le buildpack
- Image native :
 - Construite avec GraalVM installé
 - Via docker



Packaging et déploiement

Jar

Docker

Images natives



jar

On peut exécuter l'application à l'aide du jar exécutable, mais le chargement des classes à partir de fichiers jar imbriqués entraîne un faible coût de démarrage.

L'exécution à partir d'une structure éclatée est plus rapide et recommandée en production.

```
java -Djar.mode=tools -jar my-app.jar extract  
java -jar my-app/my-app.jar
```



CDS

Le partage de données de classe (CDS) est une fonctionnalité JVM qui peut aider à réduire le temps de démarrage et l'empreinte mémoire des applications Java.

```
java -Djarmode=tools -jar my-app.jar extract --destination application  
cd application
```

```
java -XX:ArchiveClassesAtExit=application.jsa -  
    Dspring.context.exit=onRefresh -jar my-app.jar
```

Ces commandes créent un fichier `application.jsa`

Pour démarrer l'application :

```
java -XX:SharedArchiveFile=application.jsa -jar my-app.jar
```



AOT avec la JVM

Le démarrage peut être optimiser à l'aide du code d'initialisation généré par AOT.

```
mvn -Pnative package
```

Puis

```
java -Dspring.aot.enabled=true -jar myapplication.jar
```

AOT a cependant certaines restrictions :

- Le classpath est fixe et défini au moment de la construction
- Les beans définis dans votre application ne peuvent pas changer :
 - L'annotation Spring `@Profile` et la configuration spécifique au profil ont des limitations.
 - Les propriétés qui changent si un bean est créé ne sont pas prises en charge (par exemple, les propriétés `@ConditionalOnProperty` et `.enable`).



Packaging et déploiement

Jar
Docker
Images natives



Recommandations

Copier et exécuter l'uber jar tel quel dans l'image Docker présente plusieurs inconvénients

- Il vaut mieux exploser le jar
- Il vaut mieux créer plusieurs couches



Création des couches

Spring Boot prend en charge l'ajout d'un fichier fournissant une liste de couches et les parties du fichier jar qui doivent être contenues : ***layers.idx***

La liste des couches dans l'index est classée en fonction de l'ordre dans lequel les couches doivent être ajoutées à l'image Docker/OCI.

Par défaut, les couches suivantes sont prises en charge :

- dependencies
- spring-boot-loader (org/springframework/boot/loader)
- snapshot-dependencies (snapshot dependencies)
- application (Classes et ressources de l'application)



Exemple

- "dependencies":
 - BOOT-INF/lib/library1.jar
 - BOOT-INF/lib/library2.jar
- "spring-boot-loader":
 - org/springframework/boot/loader/launch/JarLauncher.class
 - ... <other classes>
- "snapshot-dependencies":
 - BOOT-INF/lib/library3-SNAPSHOT.jar
- "application":
 - META-INF/MANIFEST.MF
 - BOOT-INF/classes/a/b/C.class



Dockerfile

Lorsque le fichier jar contient un fichier d'index des couches, spring-boot-jarmode-tools est ajouté en tant que dépendance.

On peut alors lancer l'application dans un mode spécial qui permet au code de bootstrap d'extraire les calques par exemple.

Cet outil peut être utilisé dans le Dockerfile



Exemple Dockerfile

```
# Perform the extraction in a separate builder container
FROM bellsoft/liberica-openjre-debian:17-cds AS builder
WORKDIR /builder
# This points to the built jar file in the target folder
# Adjust this to 'build/libs/*.jar' if you're using Gradle
ARG JAR_FILE=target/*.jar
# Copy the jar file to the working directory and rename it to application.jar
COPY ${JAR_FILE} application.jar
# Extract the jar file using an efficient layout
RUN java -Djarmode=tools -jar application.jar extract --layers --destination extracted

# This is the runtime container
FROM bellsoft/liberica-openjre-debian:17-cds
WORKDIR /application
# Copy the extracted jar contents from the builder container into the working directory in the runtime container
# Every copy step creates a new docker layer
# This allows docker to only pull the changes it really needs
COPY --from=builder /builder/extracted/dependencies/ ./
COPY --from=builder /builder/extracted/spring-boot-loader/ ./
COPY --from=builder /builder/extracted/snapshot-dependencies/ ./
COPY --from=builder /builder/extracted/application/ ./
# Start the application jar - this is not the uber jar used by the builder
# This jar only contains application code and references to the extracted jar files
# This layout is efficient to start up and CDS friendly
ENTRYPOINT ["java", "-jar", "application.jar"]
```



Cloud Native Buildpacks

Spring Boot prend en charge les buildpacks pour Maven et Gradle.

Buildpack supporte le fichier layers.idx

```
$ mvn -Pnative spring-boot:build-image
```

```
$ gradle bootBuildImage
```



Packaging et déploiement

Jar
Docker
Images natives



GraalVM Native

Les images natives GraalVM sont des exécutables autonomes qui peuvent être générés en utilisant AOT.

Une image native GraalVM est un exécutable complet et spécifique à la plateforme. Pas besoin de JVM



Différence avec un déploiement JVM

L'analyse statique de votre application est effectuée au moment de la construction à partir du point d'entrée principal.

Le code qui n'est pas accessible lors de la création de l'image native sera supprimé et ne fera pas partie de l'exécutable.

GraalVM n'est pas directement au courant des éléments dynamiques de votre code et doit être informé de la réflexion, des ressources, de la sérialisation et des proxys dynamiques.

Le classpath est fixé au moment de la construction et ne peut pas changer.

Il n'y a pas de chargement lazy de classe, tout ce qui est livré dans les exécutables sera chargé en mémoire au démarrage.

Il existe certaines limitations concernant certains aspects des applications Java qui ne sont pas entièrement prises en charge.



Fichiers générés

Une application traitée par Spring AOT génère généralement :

- Code source Java
- Bytecode (pour les proxys dynamiques, etc.)
- Fichiers JSON pour GraalVM dans META-INF/native-image/{groupId}/{artifactId}/ :
 - Indications de ressources (resource-config.json)
 - Indications de réflexion (reflect-config.json)
 - Indications de sérialisation (serialization-config.json)
 - Indications de proxy Java (proxy-config.json)
 - Indications JNI (jni-config.json)



Configuration embarquée

Les hints sont automatiquement créés pour les `@ConfigurationProperties` mais les propriétés de configuration imbriquées doivent être annotées avec **`@NestedConfigurationProperty`**.

```
@ConfigurationProperties(prefix = "my.properties")
public class MyProperties {

    private String name;

    @NestedConfigurationProperty
    private final Nested nested = new Nested();

    ...
}
```



Runtime hints

GraalVM doit savoir à l'avance si un composant utilise la réflexion ou si il doit charger une ressource du classpath

L'API ***RuntimeHints*** collecte les besoins de réflexion, de chargement de ressources, de sérialisation et de proxys JDK au moment de l'exécution.

Ex :

```
runtimeHints.resources().registerPattern("config/app.properties");
```

Spring fournit plusieurs annotations permettant d'enregistrer les besoins :

- ***@ImportRuntimeHints***
- ***@RegisterReflectionForBinding***



@ImportRuntimeHints

Les implémentations de **RuntimeHintsRegistrar** permettent d'exécuter un callback vers l'instance RuntimeHints gérée par le moteur AOT

```
@Component
@ImportRuntimeHints(MyComponentRuntimeHints.class)
public class MyComponent {

    // ...

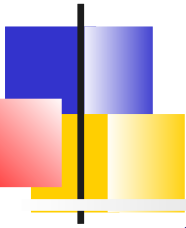
    private static class MyComponentRuntimeHints implements RuntimeHintsRegistrar {

        @Override
        public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
            // Register method for reflection
            Method method = ReflectionUtils.findMethod(MyClass.class, "sayHello", String.class);
            hints.reflection().registerMethod(method, ExecutableMode.INVOKE);

            // Register resources
            hints.resources().registerPattern("my-resource.txt");

            // Register serialization
            hints.serialization().registerType(MySerializableClass.class);

            // Register proxy
            hints.proxies().registerJdkProxy(MyInterface.class);
        }
    }
}
```



@RegisterReflectionForBinding

@RegisterReflectionForBinding est une spécialisation de *@Reflective* qui enregistre le besoin de sérialiser des types arbitraires.

UC typique est l'utilisation de DTO que le conteneur ne peut pas déduire, comme l'utilisation d'un client Web dans un corps de méthode.

```
@Component
public class OrderService {

    @RegisterReflectionForBinding(Account.class)
    public void process(Order order) {
        // ...
    }

}
```



Conversion en image native

Un fichier jar exécutable Spring Boot peut être converti en une image native à condition que le fichier jar contienne les ressources générées par AOT.

Cela peut être effectué via :

- Cloud Native Buildpacks
- l'outil d'image native fourni avec GraalVM.



Buildpack

pack peut être utilisé pour transformer un fichier exécutable Spring Boot traité par AOT en une image de conteneur native.

Installation de pack

```
pack build --builder paketobuildpacks/builder-jammy-java-tiny \
  --path target/myproject-0.0.1-SNAPSHOT.jar \
  --env 'BP_NATIVE_IMAGE=true' \
  my-application:0.0.1-SNAPSHOT
docker run --rm -p 8080:8080
  docker.io/library/myproject:0.0.1-SNAPSHOT
```



SpringBoot

Spring Initializer permet de simplifier le processus.

Maven

- Vérifier la section *<parent>*
spring-boot-starter-parent
- *native-maven-plugin* dans les build plugins

```
$ mvn -Pnative spring-boot:build-image
```

Gradle

- Vérifier la présence du plugin
org.graalvm.buildtools.native

```
$ gradle bootBuildImage
```



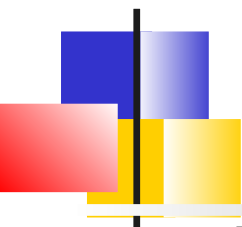
GraalVM Native images

GraalVM doit être installé

- GRAALVM_HOME doit être positionné
- GRAALVM_HOME/bin est dans le PATH

Le compilateur natif native-image spécifique à la plateforme doit être présent

```
$ rm -rf target/native
$ mkdir -p target/native
$ cd target/native
$ jar -xvf ../myproject-0.0.1-SNAPSHOT.jar
$ native-image -H:Name=myproject
@META-INF/native-image/argfile -cp .:BOOT-INF/classes:`find
BOOT-INF/lib | tr '\n' ':'`
$ mv myproject ../
```



Tracing Agent

Le ***tracing agent*** de GraalVM permet d'intercepter la réflexion, les ressources ou l'utilisation du proxy sur la JVM afin de générer les hints associés.

Il peut être utilisé pour identifier les entrées loupées par Spring

2 techniques pour la mise en place :

- Lancez l'application et l'utiliser
- Exécutez des tests d'application pour utiliser l'application.



Usage

```
java -Dspring.aot.enabled=true \  
-agentlib:native-image-agent=config-output-dir=/path/to/config-dir/ \  
-jar target/myproject-0.0.1-SNAPSHOT.jar
```

Utiliser l'application puis Ctrl+C

L'agent écrit les fichiers de hints dans le répertoire de sortie spécifié.

Pour les utiliser, les copier dans le répertoire ***src/main/resources/META-INF/native-image/***.

Lors de la création de l'image native, GraalVM prendra en compte ces fichiers de hints



Spring Boot

Maven

- Même vérifications

```
$ mvn -Pnative native:compile
```

Gradle

- Même vérifications

```
$ gradle nativeCompile
```

Le projet peut ensuite être démarré via :

```
$ target/myproject
```



Merci!!!

❖ MERCI DE VOTRE ATTENTION



Annexes

Rappels OpenID / OAuth2



Rôles du protocole

Le **client** est l'application qui essaie d'accéder à des ressources protégées

- Elle a besoin d'obtenir le consentement d'un utilisateur autorisé
=> Elle accède aux infos de l'utilisateur.

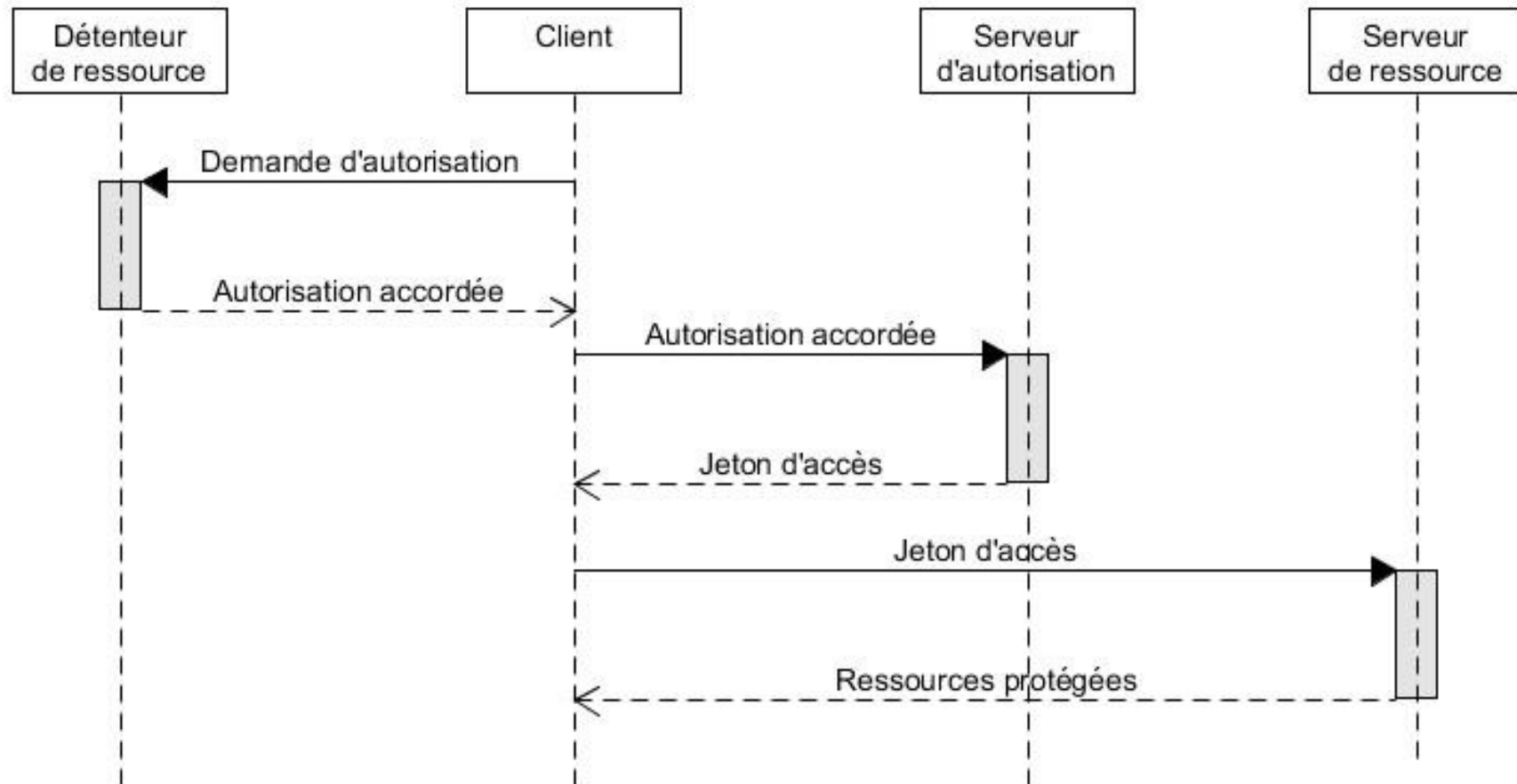
Le **serveur de ressources** est l'API utilisée pour accéder aux ressources protégées

Le **serveur d'autorisation** est le serveur fournissant un jeton.
Différents flows sont possibles pour obtenir le jeton : les grant types

L'utilisateur est la personne qui détient les ressources protégées

Rq : Un participant du protocole peut jouer plusieurs rôles

Séquence





Mise en place

S'équiper d'un serveur d'autorisation :

- Y importer des users
- Déclarer les applications clientes : leurs grant types, leurs scopes

Client

Utiliser OpenID pour authentifier un utilisateur final et obtenir un jeton contenant ses claims.

Accéder aux ressources distantes en utilisant le token

Resource server

Valider le jeton

Convertir les claims en GrantedAuthorities

Accepter ou refuser l'accès



Jetons

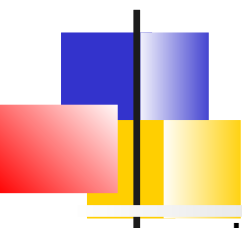
Les Tokens sont des chaînes de caractères aléatoire générées par le serveur d'autorisation

Ils sont présents dans les requêtes HTTP (entête Authorization) et contiennent des informations sensibles => HTTPS

```
GET /profile HTTP/1.1  
Host: api.example.com  
Authorization: Bearer MzJmNDc3M2VjMmQzN
```

Il y a plusieurs types de jeton délivrés par le serveur d'autorisation :

- **L'ID token** : il contient les informations utiles sur l'utilisateur
- Le **jeton d'accès** : Il a une durée de vie limitée. Il contient les informations permettant de déduire les ACLs
- Le **Refresh Token** : Délivré avec le jeton d'accès. Il est renvoyé au serveur d'autorisation pour renouveler le jeton d'accès lorsque celui-ci a expiré



Contenu du jeton et scopes

Le contenu du jeton peut être assez varié en fonction de la configuration du serveur d'autorisation et du **scope** demandé par le client.

On y trouve généralement :

- Des informations sur le serveur d'autorisation
- Des informations d'identification de l'utilisateur (login, email, ...)
- Des informations sur le rôle des utilisateur permettant une stratégie d'autorisation RBAC
- Des informations sur les scopes demandés par le client

Le scope représente donc un ensemble de données partagées avec le client

Les scopes accessibles par le client sont définis par le serveur d'autorisation.

Le scope **openid** et le scope nécessaire pour l'authentification via OpenID



OAuth2 grant types

Différents moyens afin que l'utilisateur donne son accord : les grant types

authorization code :

- 1) L'utilisateur est dirigé vers le serveur d'autorisation
- 2) L'utilisateur consent sur le serveur d'autorisation
- 3) Le serveur d'autorisation fournit un code d'autorisation via une URL de redirection
- 4) Le client utilise le code pour obtenir le jeton

implicit : Jeton fourni directement. Certains serveurs interdisent de mode

password : Le client fournit les crédeniels de l'utilisateur. Pas recommandé

client credentials : Le client est l'utilisateur, ses crédeniels suffisent à obtenir un jeton

device code : Mode utilisé lorsque il n'y pas de navigateur disponible sur le client



Validation du jeton

Lors de la réception du jeton, le serveur de ressource doit valider l'authenticité du jeton et extraire ses informations différentes techniques sont possibles

- Appel REST vers le serveur d'autorisation
- Utilisation de JWT (Json Web Token) et validation via clé privé ou clé publique

Le format JWT est recommandé car il permet d'économiser un aller/retour vers le serveur d'autorisation.

Outre son format facilement parsable, JWT permet de garantir l'authenticité du jeton (le jeton n'a pas été généré par un site malveillant)

Il existe 2 types de jetons JWT :

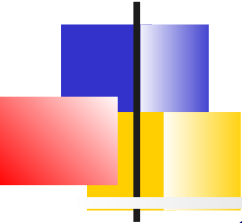
- Jeton transparent : Les données concernant l'utilisateur sont visibles
- Jeton opaque : Les données sont cryptées et nécessitent une clé supplémentaire pour les décrypter



JOSE

JWT est issu d'une famille de spécifications connue sous le nom de **JOSE**

- **JSON Web Token (JWT, RFC 7519)** : Le jeton se compose de 2 documents JSON encodés en base64 et séparés par un point : un en-tête et un ensemble de revendications (claims)
- **JSON Web Signature (JWS, RFC 7515)** : Ajoute une signature numérique de l'en-tête et des revendications
- **JSON Web Encryption (JWE, RFC 7516)** : Chiffre les revendications
- **JSON Web Algorithms (JWA, RFC 7518)** : Définit les algorithmes cryptographiques qui doivent être utilisés pour JWS et JWE
- **JSON Web Key (JWK, RFC 7517)** : Définit un format pour représenter les clés cryptographiques au format JSON



OpenID Connect

OpenID Connect s'appuie sur OAuth 2.0 pour ajouter une couche d'authentification

Il apporte :

- Social login, (se logger avec son compte Google)
- SSO dans le cadre d'une entreprise
- Les applications clientes n'ont pas accès aux mots de passe des utilisateurs

Il est compatible avec des mécanismes d'authentification forte comme le OTP (One Time Password) ou WebAuthn

OpenID Connect spécifie clairement le format JWT comme format du jeton

Flow OpenID

