GOIB VICEPRESIDÈNCIA I CONSELLERIA INNOVACIÓ, RECERCA I TURISME
DIRECCIÓ GENERAL DESENVOLUPAMENT TECNOLÒGIC

una manera de hacer europa

Fondo Europeo de Desarrollo Regional

Unió Europea

HELIUM

Arquitectura de microserveis

30 de març de 2021

Serveis d'Administració Electrònica en el Govern de les Illes Balears

Lot2 (Serveis de tramitació d'expedients electrònics)

Oficina Tècnica de Direcció de Projecte

Control de versions del document

Data	Autor	Versió	Canvis
30/03/2021	Límit tecnologies	1.0	Versió inicial

ÌNDEX

1. Introducció

Aquest document pretén ser una guia bàsica sobre l'arquitectura que s'utilitzarà en el desenvolupament dels Microserveis de Helium.

2. Diferents arquitectures en diferents entorns

S'ha de tenir en compte que podem disposar de diferents arquitectures depenent de l'entorn en que ens trobem.

Inicialment podem diferenciar els següents entorns:

- Desenvolupament microservei local
- Desenvolupament microservei en spring-cloud
- Spring-cloud en Docker-compose
- Openshift

En els projectes Spring es poden utilitzar Profiles per a indicar les configuracions que s'han d'utilitzar i les que no (o classes a carregar), i per a definir fitxers de propietats.

L'anotació utilitzada és @Profile, que accepta una llista de strings com a paràmetres. Aquests Strings poden ser noms de profiles, o bé una expressió.

En cas de ser un profile, indica que si aquell profile està actiu es tindrà en compte la classe. En cas d'una expressió es tindrà en compte la classe si es compleix l'expressió.

En les expressions es suporten els següents operants:

- ! - Operador lògic NOT
- & - Operador lògic AND
- | - Operador lògic OR

Els operadors lògics & i | no es poden mesclar sense utilitzar parèntesi.

Exemples de profiles:

- @Profile("p1", "p2") - Vàlid si el perfil P1 o el P2 estan actius.
- @Profile("p1", "!p2") - Vàlid si el perfil P1 està actiu, o el P2 no està actius.
- @Profile("p1 & p2") - Vàlid si el perfil P1 i el P2 estan actius.
- @Profile("p1 | p2") - Vàlid si el perfil P1 o el P2 estan actius.
- @Profile("p1 | (!p2 & p3") - Vàlid si el perfil P1 està actiu o el P2 està inactiu i el p3 actiu.

Els fitxers de propietats també es poden agafar per profile.

Així el fitxer application.properties (o application.yml) és el fitxer global de propietats de l'aplicació. Ara bé, si es defineixen altres fitxers de propietats amb el nom application-<PROFILE>.properties, si el profile es troba actiu, llavors les propietats definides en aquest fitxer s'ajunten amb les definides al fitxer de propietats globals, i en cas d'haver-hi propietats amb el mateix nom, tenen prioritat les definides en el fitxer de propietats del profile.

Per exemple, si tenim 2 fitxers de propietats:

application.properties

spring.application.name=app-name
server.port=8080

application-perfil.properties

server.port=8082
una.propietat.perfil=valor

i el profile "perfil" està actiu, les propietats resultants serien:

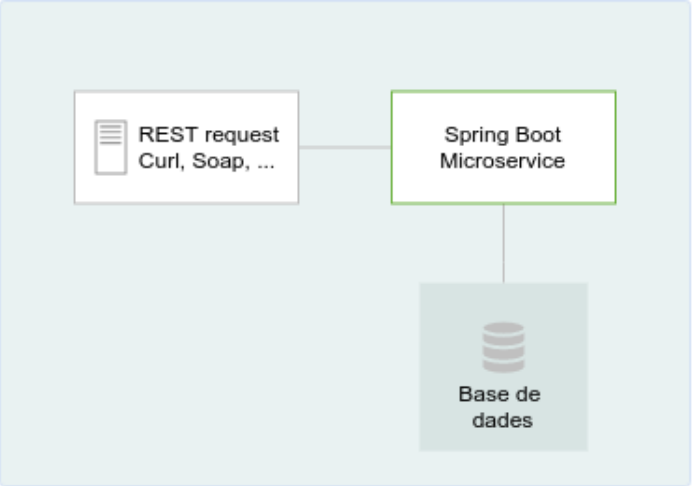
pring.application.name=app-name
server.port=8082
una.propietat.perfil=valor

La propietat server.port=8080 és sobreescrita per la de les propietats del profile "perfil".

2.1. DESENVOLUPAMENT DE MICROSERVEI EN LOCAL

En aquest cas, el que es vol és utilitzar el mínim consum de memòria possible, per tal de no perjudicar el rendiment de la màquina de treball.

Per això s'utilitzarà el Profile "**local**", i només s'executarà el microservei que s'estigui desenvolupant.



El fitxer de propietats haurà de tenir totes les propietats necessàries per al funcionament del microservei.

També haurà d'incloure les propietats necessàries per a desactivar els serveis cloud, com la integració amb els microserveis de registre, configuració i logs.

Exemple de fitxer de propietats:

application-local.properties

```
# Dades de l'aplicació
spring.application.name=aplicacio-nom
server.port=8082

# Desectivar serveis cloud
spring.zipkin.enabled=false
spring.cloud.discovery.enabled=false

# Dades de connexió a OracleDB
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@//database-host:1521/xe
spring.datasource.username=db-name
```

```
spring.datasource.password=db-pass

# Configuració HikariCP
spring.datasource.hikari.minimumIdle=5
spring.datasource.hikari.maximumPoolSize=20
spring.datasource.hikari.idleTimeout=30000
spring.datasource.hikari.maxLifetime=2000000
spring.datasource.hikari.connectionTimeout=30000
spring.datasource.hikari.poolName=HikariPoolName

# Configuració JPA
spring.jpa.database=oracle
spring.jpa.database-platform=org.hibernate.dialect.Oracle12cDialect
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=false
spring.jpa.properties.hibernate.format_sql=false

# Configuració OpenAPI
springdoc.swagger-ui.url=/openapi.yaml
springdoc.swagger-ui.path=/api/documentation.html

# Evitar que s'envii la traça de l'error en les respostes REST
server.error.include-stacktrace=never
```

2.2. DESENVOLUPAMENT DE MICROSERVEI EN SPRING-CLOUD

En aquest ca el que es vol és executar el mínim imprescindible dins un entorn cloud.

Per això s'utilitzarà el profile "**spring-cloud**", i serà necessari executar els següents serveis:

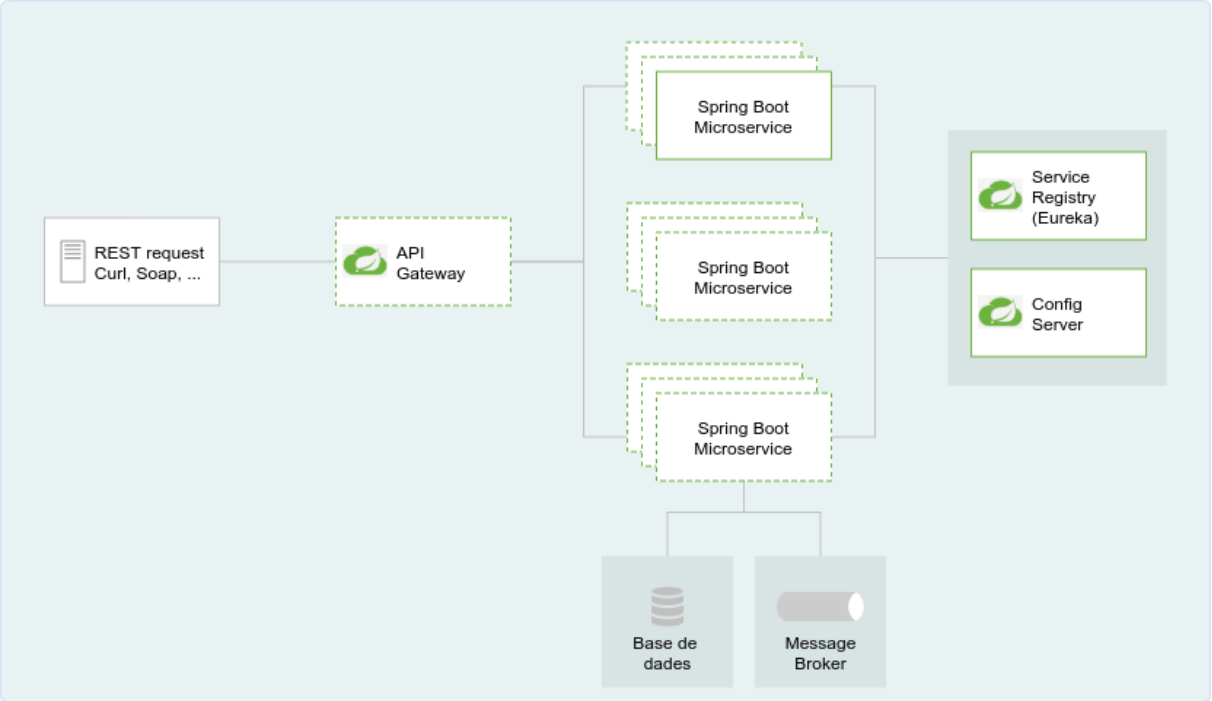
Serveis imprescindibles:

- **helium-eureka**: es tracta del Service Registry del sistema, allà on es registren totes les aplicacions del sistema.
- **helium-config-server**: allà on es troben emmagatzemades les configuracions de cada una de les aplicacions del sistema
- microservei que s'estigui desenvolupant-

Serveis opcionals:

- **helium-gateway**: en el cas que es vulgui accedir al microservei que s'està desenvolupant a través del gateway del sistema.

- altres microserveis que cridin, o siguin cridats per el microservei desenvolupat. Aquests microserveis sempre son sustituibles per eines com postman o wiremock.



És important tenir en compte que al utilitzar aquesta configuració s'utilitzaran els fitxers de configuració del config server.

Configuracions necessàries:

És necessari configurar l'aplicació per a que utilitzi el Service Registry (Eureka). Per això és necessari infloure una classe de configuració com la següent:

```
@Profile(value = {"spring-cloud"})
@EnableDiscoveryClient
@Configuration
public class LocalDiscovery {}
```

Propietats necessàries

- bootstrap.properties. I més concretament el fitxer **bootstrap-spring-cloud.properties**, ja que s'executa amb el profile "spring-cloud". Aquest fitxer es carrega prèviament al fitxer de propietats del microservei (application.properties), i és allà on es configurarà l'accés al Service Registry (Eureka), i al servidor de configuracions:

```
# Habilitar obtenció de configuracions de spring-cloud-config
spring.cloud.config.discovery.enabled=true

# Nom i credencials del servidor de configuracions
spring.cloud.config.discovery.service-id=helium-config-service
spring.cloud.config.username=USER
spring.cloud.config.password=PASS

# En cas d'errada al obtenir configuracions l'aplicació s'aturarà
spring.cloud.config.fail-fast=true

# Configuració per a connectar-se al Service Registry
eureka.client.service-url.defaultZone=http://USER:PASS@EUREKA-HOST:8761/eureka
```

- **application.properties**. És necessari configurar el nom del microservei:

```
# Nom del microservei
spring.application.name=helium-MICROSERVEI-service
```

La resta de configuracions es trobaran en el servidor de configuracions. Concretament ens trobarem amb 4 fitxers:

- **application.properties**. Configuracions globals de tots els microserveis de Helium.

```
# Configuració OracleDB
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

# Configuració HikariCP
spring.datasource.hikari.minimumIdle=5
```

```
spring.datasource.hikari.maximumPoolSize=20
spring.datasource.hikari.idleTimeout=30000
spring.datasource.hikari.maxLifetime=2000000
spring.datasource.hikari.connectionTimeout=30000
spring.datasource.hikari.poolName=HikariPoolDominis

# Configuració JPA
spring.jpa.database=oracle
spring.jpa.database-platform=org.hibernate.dialect.Oracle12cDialect
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=false
spring.jpa.properties.hibernate.format_sql=false

# Evitar que s'envii la traça de l'error en les respostes REST
server.error.include-stacktrace=never
```

- **application-spring-cloud.properties**. Configuracions globals de tots els microserveis de Helium, que s'utilitzaran al executar-se amb el profile "spring-cloud" actiu.
  Aquestes propietats complementen, i sobreescriuen, en cas d'existir, les del fitxer global application.properties.

```
# Habilitar serveis cloud i configuracions
spring.cloud.discovery.enabled=true
spring.cloud.config.discovery.service-id=helium-config-service
eureka.instance.prefer-ip-address=true

# Habilitar Logs a Zipkin (configuració Cloud)
spring.zipkin.enabled=true
spring.zipkin.base-url=http://ZIPKIN-HOST:9411/
```

- **helium-MICROSERVEI-service/application.properties**.
  Configuracions particulars del microservei.
  Aquestes propietats complementen, i sobreescriuen, en cas d'existir, les globals.

```
# Dades de l'aplicació
spring.application.name=helium-MICROSERVEI-service
server.port=8082

# Configuració OpenAPI
springdoc.swagger-ui.url=/openapi.yaml
springdoc.swagger-ui.path=/api/documentation.html

# Deshabilitar Logs a Zipkin (configuració No Cloud)
spring.zipkin.enabled=false
```

- **helium-MICROSERVEI-service/application-spring-cloud.properties**.

  Configuracions del microservei, que s'utilitzaran al executar-se amb el profile "spring-cloud" actiu.

  Aquestes propietats complementen, i sobreescriuen, en cas d'existir, les globals i les del fitxer helium-MICROSERVEI-service/application.properties.

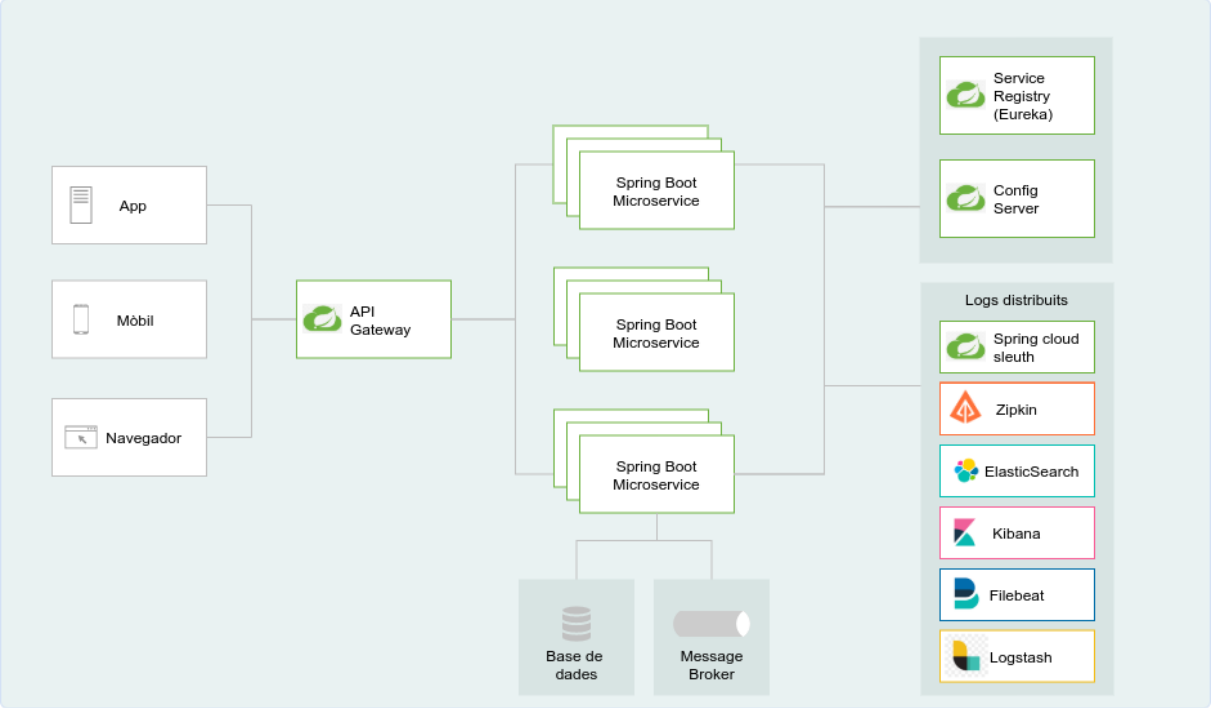
```
# Configuració OracleDB
spring.datasource.url=jdbc:localhost:thin:@//oracle:1521/xe
spring.datasource.username=USER
spring.datasource.password={cipher}PASS_ENCRIPTAT

# Habilitar Logs a Zipkin (configuració Cloud)
spring.zipkin.enabled=true
spring.zipkin.base-url=http://localhost:9411/
```

2.3. SPRING-CLOUD EN DOCKER-COMPOSE

Aquesta és la configuració a utilitzar quan es volen arrancar tots els serveis de Helium en un entorn on no es disposi de Openshift.

Per aquesta arquitectura s'utilitzarà el profile "compose".



Les configuracions faran ús del servidor de configuracions, i per tant seran molt similars a les utilitzades en el cas anterior.

Configuracions necessàries:

És necessari configurar l'aplicació per a que utilitzi el Service Registry (Eureka). Per això és necessari infloure una classe de configuració com la següent:

```
@Profile(value = {"compose"})
@EnableDiscoveryClient
@Configuration
public class LocalDiscovery {}
```

Configuracions necessàries:

- bootstrap.properties. I més concretament el fitxer **bootstrap-compose.properties**, ja que s'executa amb el profile "compose". Aquest fitxer es carrega prèviament al fitxer de propietats del microservei (application.properties), i és allà on es configurarà l'accés al Service Registry (Eureka), i al servidor de configuracions:

```
# Habilitar obtenció de configuracions de spring-cloud-config
spring.cloud.config.discovery.enabled=true

# Nom i credencials del servidor de configuracions
spring.cloud.config.discovery.service-id=helium-config-service
spring.cloud.config.username=USER
spring.cloud.config.password=PASS

# En cas d'errada al obtenir configuracions l'aplicació s'aturarà
spring.cloud.config.fail-fast=true

# Configuració per a connectar-se al Service Registry
eureka.client.service-url.defaultZone=http://USER:PASS@EUREKA-HOST:8761/eureka
eureka.instance.prefer-ip-address=true
```

- **application.properties**. És necessari configurar el nom del microservei:

```
# Nom del microservei
spring.application.name=helium-MICROSERVEI-service
```

La resta de configuracions es trobaran en el servidor de configuracions. Concretament ens trobarem amb 4 fitxers:

- **application.properties**. Configuracions globals de tots els microserveis de Helium.

```
# Configuració OracleDB
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
```

```
# Configuració HikariCP
spring.datasource.hikari.minimumIdle=5
spring.datasource.hikari.maximumPoolSize=20
spring.datasource.hikari.idleTimeout=30000
spring.datasource.hikari.maxLifetime=2000000
spring.datasource.hikari.connectionTimeout=30000
spring.datasource.hikari.poolName=HikariPoolDominis

# Configuració JPA
spring.jpa.database=oracle
spring.jpa.database-platform=org.hibernate.dialect.Oracle12cDialect
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=false
spring.jpa.properties.hibernate.format_sql=false

# Evitar que s'envii la traça de l'error en les respostes REST
server.error.include-stacktrace=never
```

- **application-compose.properties**. Configuracions globals de tots els microserveis de Helium, que s'utilitzaran al executar-se amb el profile "spring-cloud" actiu. Aquestes propietats complementen, i sobreescriuen, en cas d'existir, les del fitxer global application.properties.

```
# Habilitar serveis cloud i configuracions
spring.cloud.discovery.enabled=true
spring.cloud.config.discovery.service-id=helium-config-service

eureka.instance.prefer-ip-address=true

# Habilitar Logs a Zipkin (configuració Cloud)
spring.zipkin.enabled=true
spring.zipkin.base-url=http://zipkin:9411/
```

- **helium-MICROSERVEI-service/application.properties**. Configuracions particulars del microservei.

Aquestes propietats complementen, i sobreescriuen, en cas d'existir, les globals.

```
# Dades de l'aplicació
spring.application.name=helium-MICROSERVEI-service
server.port=8082

# Configuració OpenAPI
springdoc.swagger-ui.url=/openapi.yaml
springdoc.swagger-ui.path=/api/documentation.html

# Deshabilitar Logs a Zipkin (configuració No Cloud)
spring.zipkin.enabled=false
```

- **helium-MICROSERVEI-service/application-compose.properties**. Configuracions del microservei, que s'utilitzaran al executar-se amb el profile "spring-cloud" actiu. Aquestes propietats complementen, i sobreescriuen, en cas d'existir, les globals i les del fitxer helium-MICROSERVEI-service/application.properties.

```
# Configuració OracleDB
spring.datasource.url=jdbc:oracle:thin:@//oracle:1521/xe
spring.datasource.username=USER
spring.datasource.password={cipher}PASS_ENCRIPTAT

# Habilitar Logs a Zipkin (configuració Cloud)
spring.zipkin.enabled=true
spring.zipkin.base-url=http://zipkin:9411/
```

2.4. OPENSHIFT

Pendent de definir.

3. Dependències Maven

El projecte fa ús d'un BOM (Bill of materials) de Maven.

Concretament es defineix un projecte maven helium-parent-bom que el que fa és definir les llibreries bàsiques que s'utilitzaran els microserveis, i 2 projectes maven fills que hereten del pare:

- **helium-mvc-service-bom**: afegeix les llibreries necessàries per a desenvolupar un microservei utilitzant spring-mvc.
- **helium-flux-service-bom**: afegeix les llibreries necessàries per a desenvolupar un microservei utilitzant spring-webflux.

Tots els microserveis d'Helium utilitzaran un d'aquests 2 projectes com a pare del projecte, de manera que ja es disposi de la majoria de llibreries necessàries per al desnvolupament.

Per a utilitzar els projectes esmentats com a pares, el que s'ha de fer és, al crear un mou microservei, definir-lo amb l'etiqueta parent:

```
<parent>
	<groupId>es.caib.helium</groupId>
	<artifactId>helium-mvc-service-bom</artifactId>
	<version>1.0</version>
</parent>
```

Les principals dependències que incorporen aquests projectes són les següents:

helium-parent-bom:

- spring-boot
- spring-statemachine
- lombok
- mapstruct
- rsql
- llibreries de test:
  - awaitility
  - spock

    - wiremock
    - mockmvc
- plugins:
    - docker (generar imatge a partir de Dockerfile)
    - test:
        - surefire i failsafe
        - jacoco per a coverage

helium-mvc-service-bom:

- spring-web
- spring-data-jpa
- spring-data-rest
- eureka
- spring-cloud-config
- zipkin + sleuth
- openfeign
- openapi

helium-flux-service-bom:

- spring-webflux
- spring-data-jpa
- eureka
- spring-cloud-config
- zipkin + sleuth

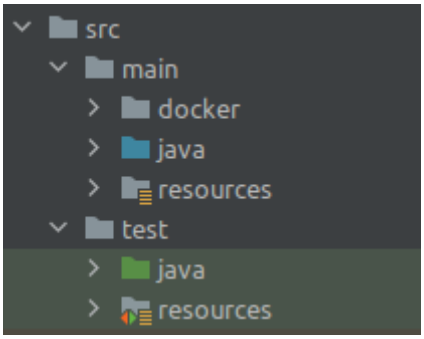
4. Estructura del microserveis

Un microservei és una aplicació que és responsable d'una funcionalitat molt concreta i limitada. Ha de fer una única feina, però l'ha de fer molt bé.

Així, normalment els microserveis defineixen molt poques entitats, serveis i controladors. De fet normalment només tindran un únic controlador rest, i un únic servei. A més, és desitjable que els microserveis siguin aplicacions el més lleugeres i ràpides possible.

Per això, donat que els microserveis d'Helium no disposen d'interfície web, enlloc de seguir l'estàndard d'aplicacions de la caib es considera més adequat crear una aplicació maven monomòdul, on es diferenciaran els diferents elements per paquests.

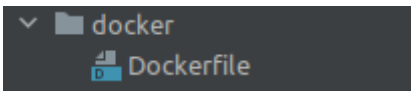
Així l'estructura bàsica d'un microservei a Helium és la següent:



Main

Pel que fa a la carpeta main, hi trobam 3 carpetes filles: docker, java i resources.

La carpeta **docker** contindrà el fitxer Dockerfile que defineix com s'ha de crear la imatge de docker del microservei.



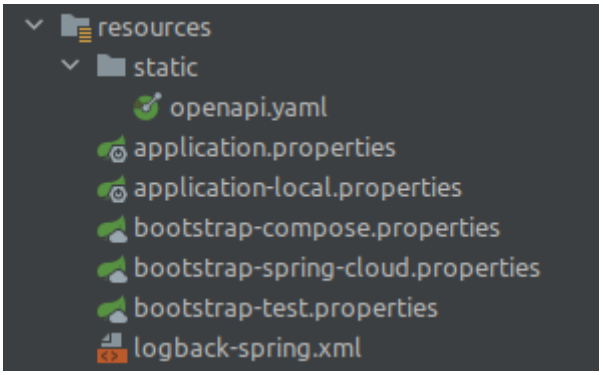
La carpeta **resources** contindrà:

- Una carpeta static amb la especificació de la API REST del microservei en format OpenAPI (fitxer yaml o json).
- Els fitxers de configuració de l'aplicació:

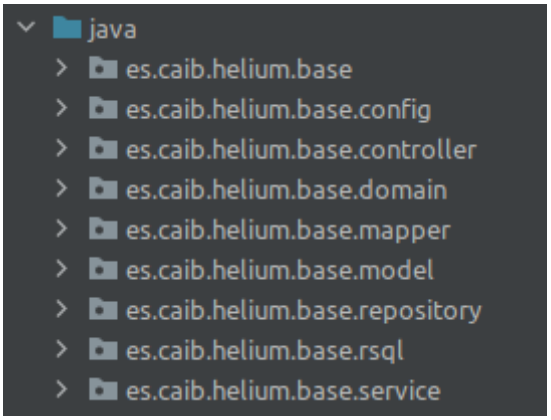
- bootstrap.properties
- applicacion.properties

i fitxers de configuració per profile bootstrap-PROFILE.properties i application-PROFILE.properties.

- Fitxer logback-spring.xml. Fitxer que defineix com s'han de convdertir els logs de l'aplicació a json, per tal que es puguin desar corredtament a zipkin i elasticsearch.



La carpeta **java** contindrà el codi font de l'aplicació. El codi font s'estructurarà en paquets, tots els quals començaran per es.caib.helium:



El contingut dels diferents paquets del projecte serà el següent:

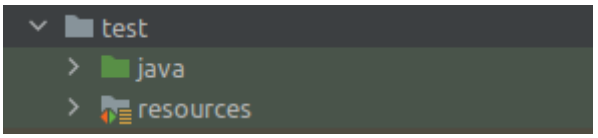
- El paquet base contindrà el fitxer que inicia el microservei
- **Config**: contindrà la configuració de l'aplicació. Concretament seran classes decorades amb la anotació @Configuration d'Spring, que defineixen totes els configuracions via java de l'aplicació.

Aquestes classes de configuració també poden anar decorades amb la anotació @Profile per a indicar amb quin profile s'han de tenir en compte i amb quin no.

- **Domain**: contindrà les entitats de l'aplicació. En general seran entitats definides amb hibernate.
- **Repository**: contindrà els repositoris per a accedir a les entitats de l'aplicació. En general seran repositoris d'spring-data.
- **Model**: contindrà els DTOs necessaris per a la API REST de l'aplicació. En general la conversió d'entitat a DTO es realitzarà als serveis.
- **Mapper**: contindrà les interfícies que defineixen les conversions entre entitat i DTOs. En general es tracta d'interfícies per a la llibreria MapStruct.
- **Service**: contindrà els serveis de l'aplicació
- **Rsql**: contindrà les classes necessàries per a parsejar un string amb format RSQL a una especificació utilitzable per els repositoris s'spring-data.
- **Controller**: contindrà els controladors REST que implementen la API REST de l'aplicació.

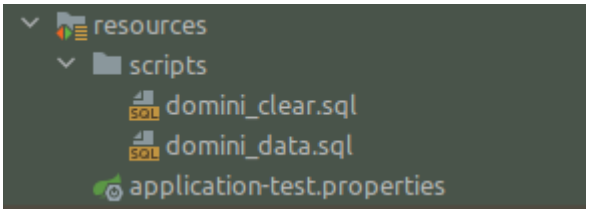
Test

En la carpeta test del projecte és on trobarem tot el referent als jocs de proves de l'aplicació.

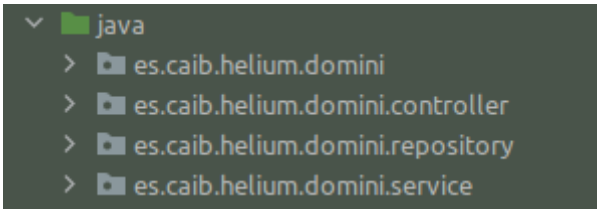


En la carpeta resources bàsicament trobarem:

- Fitxers de propietats per a la inicialització del context d'Spring en mode test (base de dades en memòria, propietats específiques de test, ...)
- Altres fitxers que puguin ser necessaris per a la realització d'alguna prova. P. ex.:
    - scripts d'inicialització de dades
    - fitxers per a enviar o rebre
    - esquemes per a wiremock



En la carpeta java és on es trobarà la implementació de les proves. Per defecte s'estructurarà en paquests com en el codi font, començant per es.caib.helium:



El contingut dels diferents paquets de les proves serà el següent:

- Carpeta base: Classes d'ajuda per les proves. P. ex.:
  - mètodes per a inicialitzar dades
  - mètodes per a realitzar comprovacions
- Controller: contindrà els jocs de proves dels controladors.
- Service: contindrà els jocs de proves dels serveis.
- Repository: contindrà els jocs de proves dels repositoris.

5. UTILITATS DESENVOLUPADES PER ALS MICROSERVEIS

En els microserveis es poden utilitzar una sèrie de llibreries o utilitats per a facilitar el desenvolupament de les APIs REST, que es troben ja en el projecte base (helium-basems).

- RSQL i Specifications
- RestTemplate
- Openfeign
- Cache distribuïda

5.1. RSQL I SPECIFICATIONS

En el paquet rsql es troben les classes necessàries per a parsejar un string en format rsql i convertir-lo en una especificació, utilitzable per el mètode findAll del respositori d'Spring-Data.

Per a poder utilitzar-ho, és necessari afegir la següent dependència maven, que ja es troba en els boms del projecte:

```
<dependency>
    <groupId>cz.jirutka.rsql</groupId>
    <artifactId>rsql-parser</artifactId>
</dependency>
```

Format RSQL

RSQL és el REST API Query Language, qué bàsicament defineix una sintaxi senzilla per a poder utilitzar com a filtre en les consultes REST.

Bàsicament el llenguatge permet definir filtres utilitzant múltiples comparacions, separades per operacions lògiques.

Les operacions lògiques són les següents:

- Logical AND : **;** or **and**
- Logical OR: **,** or **or**

Les operacions de comparació disponibles són les següents:

- Igual a : **==**
- Diferent a : **!=**
- Menor que : **=lt=** or **<**
- Menor que o igual a : **=le=** or **<=**
- Major que : **=gt=** or **>**
- Major que o igual a : **=ge=** or **>=**
- En : **=in=**
- No en : **=out=**
- Igual IgnoreCase : **=ic=**
- Diferent IgnoreCase : **=nic=**
- És null : **=isnull=**
- No és null : **=notnull=**

Es pot utilitzar el caràcter * com a comodí per indicar que hi pot anar qualsevol nombre de caràcters.

Exemples:

- name=="Kill Bill";year=gt=2003
- genres=in=(sci-fi,action);genres=out=(romance,horror),director==Que*Tarantino
- director=ic=*taran*

Si es volen afegir més operacions es pot fer modificant els fitxers:

- RsqlSearchOperation.java
- GenericRsqlSpecification.java

Per més informació veure:

https://github.com/jirutka/rsql-parser,
https://www.baeldung.com/rest-api-search-language-rsql-fiql

Parsejar RSQL

El el projecte s'ha definit la classe ServiceHelper, que ofereix una sèrie de mètodes que ajudar en l'ús de l'RSQL.

Tot seguit s'expliquen els diferents mètodes disponibles:

- Specification<T> **getRsqlSpecification**(String filtreRSQL)

  Métode que parseja un string amb format RSQL i retorna la seva corresponent especificació.

- Specification<E> **joinRsqlAndSpecification**(

  Specification<E> spec,

  String filtreRsql)

  Mètode que parseja un string amb format RSQL i junta aplicant un AND amb l'especificació **spec** passada per paràmetre.

- Page<E> **getEntityPage**(

  BaseRepository<E, PK> repository,

  Specification<E> spec,

  String filtreRsql,

  Pageable pageable,

  Sort sort)

  Mètode que realitza una consulta aplicant especificacions, i retorna una pàgina.

  Paràmetres:

  - **repository**: Obligatori. Repositori que s'utilitzarà per a realitzar la consulta. Concretament utilizarà el mètode repository.findAll().

    El repository ha d'extendre la interfície BaseReposotory, o si nó, extendre el JpaRepository i JpaSpecificationExecutor.

  - **spec**: Opcional. Al realitzar la consulta es pot passar una especificació a aplicar (a part del filtre rsql).

    Si es passa una especificació i un filtre es realitzarà un AND amb les 2 especificacions, i l'especificació resultant serà la que es passarà al mètode findAll del repositori.

  - **filtreRsql**: Opcional. Cadena amb format RSQL que definirà una especificació per a filtrar la consulta.

  - **pageable**: Obligatori. Informació de paginació i ordenació que es vol aplicar a la consulta. Si no es vol paginació es pot indicar un Pageable.unpaged().

    L'objecte pageable es pot generar de forma automàtica al controlador. Per a que

es generi, els paràmetres que s'han de passar al fer la crida REST són els següents:

- page: número de la pàgina a obtenir (comença per 0)
- size: nombre d'elements de la pàgina
- sort: ordre a aplicar.

  Aquest camp pot apareixer múltiples vegades, si es vol ordenar per més d'un camp.

  El format del sort és camp,[asc|desc] (asc i desc són opcionals. Si no es posa, per defecte s'aplica la ordenació asc)

Exemple:

your/uri?**page=0&size=25&sort=name,asc&sort=title,desc**

Al controlador s'ha de posar un Pageable com a paràmetre.

Exemple:

```
public ResponseEntity<PagedList> listBasesV1(
        final Pageable pageable,
        final Sort sort) { ... }
```

- sort: Opcional. Si s'ha indicat un pageable unpaged, llavors, encara es pot indicar l'ordre amb que es voles obtenir els resultats de la consulta.

  A la consulta REST s'ha de passar un, o múltiples camps sort.

  Exemple:

  your/uri?**sort=name,asc&sort=title,desc**

  Al controlador s'ha de posar un Sort com a paràmetre.

  Exemple:

```
public ResponseEntity<PagedList> listBasesV1(
        final Pageable pageable,
        final Sort sort) { ... }
```

- Page<E> **getEntityPage**(

  BaseRepository<E, PK> repository,

  Specification<E> spec,

  String filtreRsql,

  Pageable pageable,

  Sort sort,

Class<?> classWithSort)

Mètode exactament idèntic a l'anterior, però amb una particularitat: permet definir un ordre per defecte. És a dir, que si no es passa cap ordre a través dels paràmetres pageable o sort, el sistema aplicarà l'ordre definit a la classe indicada al paràmetre classWithSort.

Per a informar de l'ordre per defecte, a la classe desitjada (per defecte el dto), s'afegirà l'anotació @DefaultSort, que contindrà una llista de @DefaultOrder.

Exemple:

```
@DefaultSort(sortFields = {
        @DefaultOrder(field = "nom", direction = Sort.Direction.DESC),
        @DefaultOrder(field = "codi", direction = Sort.Direction.ASC)
})
```

- PagedList<D> **getDtoPage**(
  BaseRepository<E, PK> repository,
  Specification<E> spec,
  String filtreRsql,
  Pageable pageable,
  Sort sort,
  Class<?> dtoClass,
  BaseMapper<E, D> mapper)

  Mètode exactament idèntic a l'anteriro, excepte per 2 diferències:
  - **dtoClass**: La classe on es pot informar l'ordre per defecte serà sempre el DTO
  - **mapper**: El llistat retornat és prèviament convertit a una llista de DTOs, utilitzant el mètode entityToDto del mapper.

Ús d'especificacions

Spring data permet realitzar consultes aplicant criterias, mitjançant especificacions.

Per aplicar-ho només cal que el repositori extengui de la interfície BaseRepository. Els repositoris que extenguin aquestes interfície disposaran dels següents mètodes:

- Optional<T> findOne(@Nullable Specification<T> var1);

- List<T> findAll(@Nullable Specification<T> var1);
- Page<T> findAll(@Nullable Specification<T> var1, Pageable var2);
- List<T> findAll(@Nullable Specification<T> var1, Sort var2);
- long count(@Nullable Specification<T> var1);

Si es volen definir especificacions es poden definir implementant el metode toPredicate de l'Specification:

```
public static Specification<Domini> belongsToEntorn(Long entorn){
        return new Specification<Product>() {
                @Override
                public Predicate toPredicate(Root<Domini> root,
                                CriteriaQuery<?> query,
                                CriteriaBuilder criteriaBuilder) {
                        return criteriaBuilder.equal(root.get("entorn"), entorn);
                 }
        };
}
```

O utilitzant expresions lambda de java8:

```
public static Specification<Domini> belongsToEntorn(Long entorn) {
        return (domini, cq, cb) -> cb.equal(domini.get("entorn"), entorn);
}
```

5.2. CLIENT REST

Spring MVC ens ofereix la classe RestTemplate que simplifica la realització de peticions a serveis REST.

Spring WebFlux ens ofereix la classe WebClient, més moderna i no bloquejant per les peticions REST.

Per més informació veure:

https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#webmvc-client

https://www.baeldung.com/rest-template

https://www.baeldung.com/spring-5-webclient

5.3. OPENFEIGN

En el projecte s'ha incorporat la llibreria OpenFeign per a simplificar les peticions entre els diferents microserveis.

Per a crear un servei que utilitzi openFeign és necessari crear una interfície i decorar-la amb l'anotació @FeignClient, a la quan se li indicarà el nom del microservei al qual realitzarà les peticions, i opcionalment una classe de FailSafe, i possibles configuracions (autenticació, ...)

A la interfície es declararan els mètodes que s'hagin d'utilitzar per de l'altre microservei amb l'anotació @RestMapping.

Exemple:

```
@FeignClient(
        name = "helium-domini-service",
        fallback = DominiFailoverService.class,
        configuration = FeignClientConfig.class)
public interface DominiFeignClient {
        String DOMINI_PATH = "/api/v1/dominis/{dominiId}";

        @RequestMapping(method = RequestMethod.GET, value = DOMINI_PATH)
        ResponseEntity<DominiDto> getDomini(@PathVariable Long dominiId);
}
```

Aquesta interfície es podrà utilitzar com si d'un mètode implementat es tractàs.

Exemple:

```
public DominiDto getDomini(Long dominiId) {
	DominiDto dominiDto = null;
	try {
		ResponseEntity<DominiDto> responseEntity =
			dominiFeignClient.getDomini(dominiId);
		...
```

5.4. CACHÉ DISTRIBUÏDA

Si es vol utilitzar caché, serà necessari que aquesta sigui distribuïda.

S'utilitzarà la caché distribuida Hazelcast, i per a poder utilitzar-la serà necessari afegir les següents dependències:

```
<dependency>
		<groupId>com.hazelcast</groupId>
		<artifactId>hazelcast</artifactId>
</dependency>
<dependency>
		<groupId>com.hazelcast</groupId>
		<artifactId>hazelcast</artifactId>
		<version>4.0.3</version>
		<classifier>tests</classifier>
		<scope>test</scope>
</dependency>
<dependency>
		<groupId>com.hazelcast</groupId>
		<artifactId>hazelcast-eureka-one</artifactId>
		<version>2.0.1</version>
</dependency>
```

També serà necessari configurar el microservei per a que les diferents intàncies d'aquest sincronitzin la caché. Així serà necessari afegir un fitxer de configuració:

```
@Configuration
public class CacheConfig {
	@Value("${hazelcast.port:5701}") // port 5701 per defecte
	private int hazelcastPort;

	@Bean
	public Config hazelcastConfig(
			@Qualifier("eurekaClient") EurekaClient eurekaClient) {

		EurekaOneDiscoveryStrategyFactory.setEurekaClient(eurekaClient);
		Config config = new Config();
		config.getNetworkConfig().setPort(hazelcastPort);
		config.getNetworkConfig().setPortAutoIncrement(true);
		config.getNetworkConfig().getJoin().getMulticastConfig()
				.setEnabled(false);
		config.getNetworkConfig().getJoin().getEurekaConfig()
				.setEnabled(true)
				.setProperty("self-registration", "true")
				.setProperty("namespace", "hazelcast")
				.setProperty("use-metadata-for-host-and-port", "true");
		return config;
	}
}
```

Per a utilitzar la caché injectarem una instància d'aquesta en el component, amb @Autowired, o com a paràmetre del constructor.

Exemple:

```
	private final HazelcastInstance hazelcastInstance;
```

Un cop configurada la caché, necessitarem un mapa distribuït per a desar els elements a la caché. Això ho podem fer declarant un simple Map, o un Imap si volem tota la funcionalitat que ofereix hazelcast.

Exemple:

```
	Map<Long, String> mapCache = hazelcastInstance.getMap("data");
	IMap<Long, String> mapCache = hazelcastInstance.getMap("data"));
```

Per desar una data a la caché només cal fer un put al mapa.

Exemple:

```
map.put(identificador, "message");
```

Per consultar el valor de la caché només cal realitzar una consulta get sobre el mapa.

Exemple:

```
map.get(identificador);
```

Si hem utilitzat un IMap tenim altres operacions disponibles com:

- putIfAbsent
- replace
- evict
- putAsync
- putTransient
- lock

Per més informació:

https://hazelcast.org/imdg/docs/

6. Jocs de proves

El desenvolupament de jocs de proves és imprescindible en el desenvolupament de microserveis. Per aquest motiu no s'acceptarà cap desenvolupament que no porti els seus corresponents jocs de proves.

Pel que fa a jocs de proves podem distingir entre jocs de proves unitaris i d'integració. A més els jocs de proves es poden fer per diferent tipus de components: controladors, serveis i repositoris.

Els jocs de proves es realitzaran amb Junit 5, i es garantirà una cobertura mínima entre el 70-80% del codi.

Per a realitzar les proves es disposa d'una sèrie de llibreries, que ja han estat incloses en el projecte:

- Mockito
- WireMock
- Harmcrest
- AssertJ
- Spock
- Awaitility
- Jacoco

Jocs de proves unitàries

Els noms de tots els fitxers amb jocs de proves unitàries acabaran amb la paraula Test.

Controladors

Es definiran amb l'anotació

```
@WebMvcTest(value = CONTROLADOR_A_TESTEAT.class)
class ControllerTest {
```

Aquesta anotació crea un context web mock per a la realització de les proves.

Si el controlador a testear fa ús d'algun servei, aquest es definirà com a mock:

```
@MockBean
ServeiService serveiService;
```

Per a poder realitzar les proves, fent ús del context web mock, serà necessari declarar-lo:

```
@Autowired
MockMvc mockMvc;
```

Aquest objecte MockMvc ofereix un mètode **perform** per a executar la petició, i un mètode **andExpect** per a comprovar el resultat obtingut.

- **perform**. En el mètode perform se li passarà un requestBuilder (una petició amb el tipus: get, post, ...).
  Ex. mockMvc.perform(get("URL_PETICIO"))
- **andExpect**. El mètode andExpect podrà utilitzar diferents objectes per a les validacions:
  - status(). Informació de l'HttpStatus de la resposta
  - header(). Informació de la capçalera de la resposta
  - model(). El model generat.
  - content(). El contingut de la resposta.
  - jsonPath(). Path per comprovar el json retornat.

Exemple:

```
@WebMvcTest(value = DominiController.class)
class DominiControllerTest {
	@MockBean
	DominiService dominiService;

	@Autowired
	MockMvc mockMvc;

	@Test
	@DisplayName("Consulta de dominis")
	void whenListDominisV1_thenReturnList() throws Exception {
		given(dominiService.listDominis(...).willReturn(...);
		mockMvc.perform(get("/api/v1/dominis").param("entornId", "1"))
				.andExpect(status().isOk())
				.andExpect(content()
```

```
                .contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.content").exists())
        .andExpect(jsonPath("$.content", hasSize(1)))
        .andExpect(jsonPath("$.content[0].nom")
                .value("Domini_nom1"));
}
```

Serveis

Per a realitzar jocs de proves unitàries a serveis utilitzarem Mockito. Així s'afegirà la següent anotació per a poder fer ús de la llibreria Mockito en les nostres proves:

```
@ExtendWith(MockitoExtension.class)
class ServeiServiceTest {
```

Tots els components injectats al servei (via injecció de dependència), que s'hagin d'utilitzar durant les proves es declararan com a mocks. Per fer això es definiran com a mock a la classe de test, i s'injectaran al servei testeat.

Ex.

```
@Mock
RepositoriRepository repositoriRepository;
@InjectMocks
ServeiServiceImpl serveiService;
```

Els test es realitzaran seguint un patró BDD (Behavior Driven Development):

Cada joc de proves te 3 parts, definides per: GIVEN → WHEN → THEN, o el que vendria a ser Precondició → Execució → Postcondició.

- GIVEN: donat unes condicions.
    - Inicialització de dades
    - Definició de respostes dels components mock
- WHEN: quan es realitza l'execució de la funcionalitat que es vol testear

- THEN: llavors s'ha de tenir el resultat esperat
  - Realització de comprovacions amb Assercions
  - Comprovació de nombre de cridades als components mock

Mockito defineix uns mètodes per seguir el potró BDD:

- Given: mètodes per a definir com es comportaran els components mock.

  Ex.

```
Domini domini = new Domini();
given(dominiService.getById(anyLong()))
        .willReturn(Optional.of(domini));

given(dominiService.getById(anyLong()))
        .willThrow(new ResponseStatusException(
                HttpStatus.NOT_FOUND,
                "Not found"));
```

- Then: mètodes per a verificar l'ús dels mètodes dels components mock.

  Ex.

```
then(dominiRepository).should(times(1)).findAll();
then(dominiRepository).should(alLeast(1)).findAll();
then(dominiRepository).shouldHaveNoMoreInteractions();
```

Jocs de proves d'integració

Els noms de tots els fitxers amb jocs de proves d'integració acabaran amb IT.

Controladors

Es definiran amb l'anotació

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@TestPropertySource(locations = "classpath:application-test.properties")
class ControllerIT {
```

Aquesta anotació inicialitza el context d'Spring, inclòs el context web, amb un port aleatori. Per a configurar el context d'Spring se li indica un fitxer de propietats on es configura una base de dades en memòria h2, enlloc de la base de dades real.

Ex.

```
spring.application.name=helium-domini-service
server.port=8082

spring.zipkin.enabled=false
spring.cloud.config.discovery.enabled=false

spring.datasource.url = jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
spring.jpa.show-sql=false
```

Es podran utilitzar tots els components de l'aplicació injectant-los en el test amb la anotació `@Autowired`:

```
@Autowired
ServeiService serveiService;
```

Per a realitzar les proves es faran crides REST als controladors. Spring ofereix un servei RestTemplate per a realitzar les peticions. Per a utilitzar-ho és necessari injectar-lo al joc de proves:

```
@Autowired
ServeiService serveiService;
```

Per a poder realitzar les proves, fent ús del context web mock, serà necessari declarar-lo:

```
@Autowired
TestRestTemplate restTemplate;
```

Un cop es deiposa del restTemplate, per a realitzar les proves es poden realitzar peticions al controlador REST, i comprovar els resultats obtinguts.

Exemple:

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@TestPropertySource(locations = "classpath:application-test.properties")
class DominiControllerIT {

	public static final String API_V1_DOMINI = "/api/v1/dominis/";

	@Autowired
	TestRestTemplate restTemplate;
	...
	@Test
	@DisplayName("Consulta de dades de domini")
	void whenListDominisV1_thenReturnList() throws Exception {
		String url = API_V1_DOMINI + "?entornId=2";
		PagedList<DominiDto> pagedList = restTemplate.getForObject(
					url,
					DominiPagedList.class);

		assertThat(pagedList.getContent()).hasSize(3);
		pagedList.getContent().forEach(dominiDto -> {
			DominiDto fetchedDominidto = restTemplate.getForObject(
				API_V1_DOMINI + dominiDto.getId(), ominiDto.class);
			assertThat(dominiDto.getId())
				.isEqualByComparingTo(fetchedDominidto.getId());
		});
	}
```

Serveis

Es definiran amb l'anotació

```
@SpringBootTest
@TestPropertySource(locations = "classpath:application-test.properties")
class ControllerIT {
```

Aquesta anotació inicialitza el context d'Spring, sense context web. Per a configurar el context d'Spring se li indica un fitxer de propietats on es configura una base de dades en memòria h2, enlloc de la base de dades real.

Ex.

```
spring.application.name=helium-domini-service
server.port=8082

spring.zipkin.enabled=false
spring.cloud.config.discovery.enabled=false

spring.datasource.url = jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
spring.jpa.show-sql=false
```

Es podran utilitzar tots els components de l'aplicació injectant-los en el test amb la anotació `@Autowired`:

```
@Autowired
ServeiService serveiService;
```

Per a realitzar les proves es faran crides als mètodes del servei, i es comprovarà el resultat. Tot i no utilitzar mockito, s'aconsella utilitzar el patró BDD per a la realització de les proves.

Ex.

```
@Test
@DisplayName("Crear domini")
void whenCreateDomini_thenReturn() {
	// Given
	Domini domini = …
	DominiDto dto = dominiMapper.entityToDto(domini);

	// When
	DominiDto creat = dominiService.createDomini(dto);

	// Then
	assertThat(creat).isNotNull();
	assertThat(creat.getId()).isNotNull();
```

```
        DominiDto trobat = dominiService.getById(creat.getId());
        comprovaDominisIguals(creat, trobat);
    }
```

Si es vol utilitzar wiremock per a crear mocks d'accés a altres microserveis, llavors també cal afegir l'anotació `@ExtendWith({WireMockExtension.class})`, i llavors injentant-lo:

```
@Managed
WireMockServer wireMockServer = with(wireMockConfig().dynamicPort());
```

L'ús del wiremock és molt semblant al de Mockito. Només cal definir com respondrà la petició a una adreça d'un servei REST (o WS).

Ex.

```
String jsonResponse = …
wireMockServer.stubFor(get(urlPathEqualTo("/api/rest/test"))
        .willReturn(okJson(jsonResponse)));
wireMockServer.stubFor(get(urlPathEqualTo("/api/rest/test"))
        .willReturn(notFound()));
```

També es disposa de mètodes per a verificar el nombre de crides als serveis realitzades, fent ús de verify.

Ex.

```
verify(1, getRequestedFor(urlPathEqualTo("/api/rest/test"))
            .withBasicAuth(new BasicCredentials("usuari", "password")));
```

Repositoris

Es definiran amb l'anotació

```
@DataJpaTest
class RepositoryIT {
```

Aquesta anotació inicialitza un context d'Spring mínim amb un entityManager, i accés a una base de dades en memòria.

Per accedir a l'entityManager és necessari injectar-lo:

```
@Autowired
private TestEntityManager entityManager;
```

Per a utilitzar el repositori a testear també cal injectar-lo:

```
@Autowired
private RepositoriRepository repositoriRepository;
```

Un cop es disposa dels 2 components, l'entity manager, i el repositori, es poden utilitzar per a realitzar les proves. S'utilitzarà el repositori per a executar la operació a provar, i l'entityManager per a inicialitzar dades, i comprovar les dades modificades pel repositori.

Exemple:

```
@DataJpaTest
class DominiRepositoryIT {
	@Autowired
	private TestEntityManager entityManager;
	@Autowired
	private DominiRepository dominiRepository;

	@Test
	@DisplayName("Consulta per entorn i id")
	void whenFindByEntornAndId_thenReturnDomini() {
		Domini creat = entityManager.persistAndFlush(domini);
		Optional<Domini> trobat = dominiRepository.findByEntornAndId(
				domini.getEntorn(),
				creat.getId());

		assertTrue(trobat.isPresent());
		comprovaDominisIguals(creat, trobat.get());
```