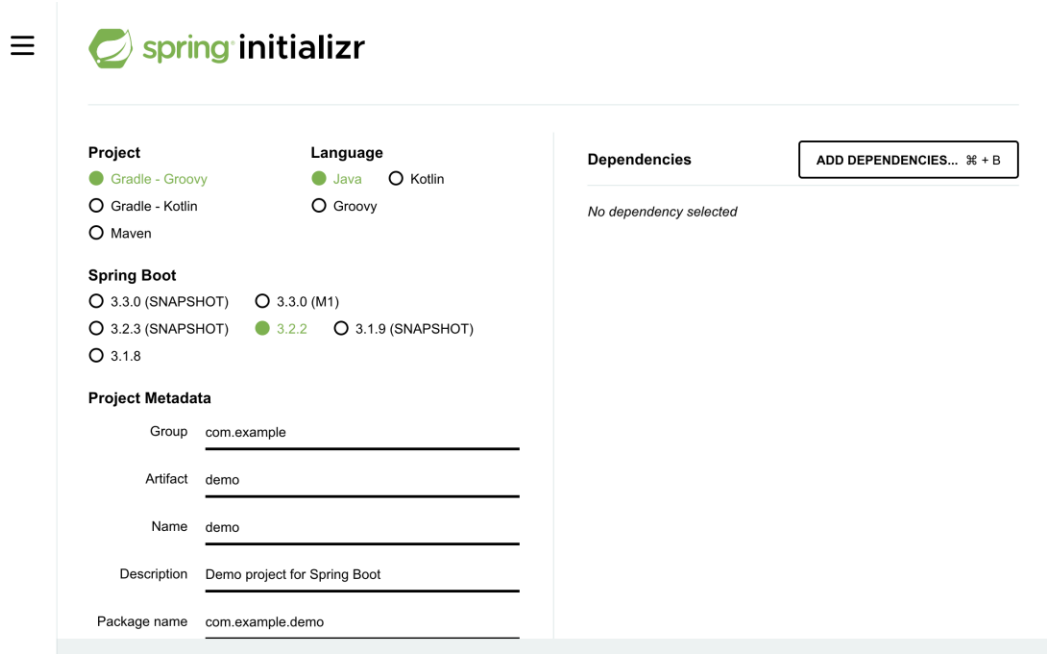
SOA
Restful Web services
Springboot

Framework Springboot

* « Micro » framework specialised to routing, Rest APIs and web apps.
* Based upon Spring (heavy MVC java framwork)
* Uses annotations (very similar to Jersey)
* Simplifies configurations (repository, logs, etc etc.)
* Supports Hateos

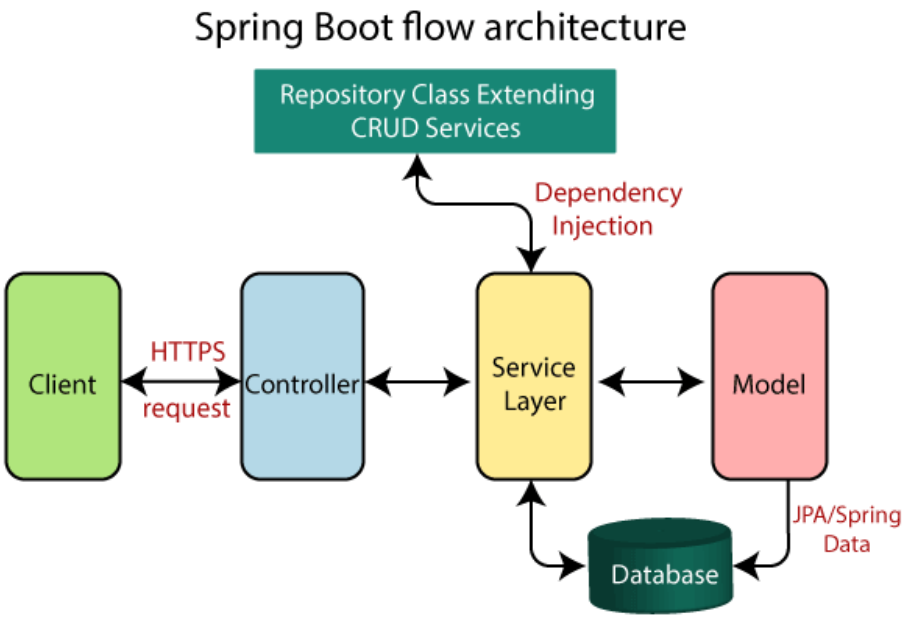
Framework Springboot

* Start a project ?
  * https://start.spring.io/ springboot initializer



Framework Springboot

* Sprinboot application architecture



Framework Springboot

* Sprinboot application architecture
* The *@Controller* annotation indicates that a particular class serves the role of a controller.
  * Web api here
* *@Services* holds business logic and calls methods in repository layer.
  * used to indicate that it's holding the business logic,
  * Not often used, for visibility ?

Framework Springboot

* Application Root :

```
@SpringBootApplication
public class HelloWorldConfiguration { public static void
main(String[] args) {
SpringApplication.run(HelloWorldConfiguration.class, args);
} }
```

SpringApplication.run() ->launch the app, searches for controllers and starts them

Framework Springboot

* Simple Business class (classe métier):

```
package hello;

public class Greeting {
private final long id;
private final String content;

public Greeting() {}
public Greeting(long id, String content) {
this.id = id;
this.content = content; }

public long getId() { return id; }
public String getContent() { return content; } }
```

Often, it 's better to add a default constructor

Framework Springboot

* A controller :

```
package hello;

import …
@Controller
@RequestMapping("/hello-world")
public class HelloWorldController {
private static final String template = "Hello, %s!";
private final AtomicLong counter = new AtomicLong();

@RequestMapping(method=RequestMethod.GET)
 public @ResponseBody Greeting sayHello(@RequestParam(value="name",
required=false, defaultValue="Stranger") String name) {
 return new Greeting(counter.incrementAndGet(), String.format(template, name)); }
 }
```

Framework Springboot

* Previous controller sends data serialised with JSON by default

* Controler persists ! (Where is the stateless principle ???)

Framework Springboot

Annotation Mapping

@ResponseBody maps the HttpRequest body to a Java Object (enabling automatic deserialization)

Setting the Content Type :

produces = MediaType.APPLICATION_JSON_VALUE

produces = MediaType.APPLICATION_XML_VALUE

Framework Springboot

Annotation Mapping

@RequestMapping

```
@RequestMapping(method=RequestMethod.GET)
 public @ResponseBody Greeting sayHello(@RequestParam(value="name",
required=false, defaultValue="Stranger") String name)
```

```
@RequestMapping(method=RequestMethod.GET, produces =
MediaType.APPLICATION_JSON_VALUE)
@ResponseBody…
```

Framework Springboot

Annotation Mapping

@RequestMapping

```
@RequestMapping(value = "/getallbooks", method =
RequestMethod.GET, produces = "application/json")
```

Framework Springboot

Annotation Mapping (URLS)

* Add some header

@RequestMapping(value = "/ex/foos", method = GET, headers = "Accept=application/json")

@ResponseBody public String getFoosAsJsonFromBrowser() { return "Get some Foos with Header Old"; }

Framework Springboot

Annotation Mapping

@RequestMapping

But also:

@GetMapping
@PostMapping
@PutMapping
@DeleteMapping
@PatchMapping

Framework Springboot

Annotation Mapping

@PathVariable

@GetMapping("/students/{studentId}/courses/{courseId}") public Course retrieveDetailsForCourse(@PathVariable String studentId, @PathVariable String courseId) {

@PathVariable With Regex

@RequestMapping(value = "/ex/bars/{numericId:[\\d]+}", method = GET)
@ResponseBody

http://localhost:8080/spring-rest/ex/bars/1 works
http://localhost:8080/spring-rest/ex/bars/abc ko

Framework Springboot

Annotation Mapping

@RequestMapping

@RequestMapping(value = "/students/", method = GET) @ResponseBody
public Course getStudentBySimplePathWithRequestParam(@RequestParam("id") long id) {
return Course}

http://localhost:8080/students/?id=100

Multiple path

@RequestMapping(value = { "/ex/advanced/bars", "/ex/advanced/foos" }, method = GET)
@ResponseBody
public String getFoosOrBarsByPath() {

Also muliple verbs;

Framework Springboot

Error Management

Build your own exception

exception translated into HTTP responses with status

Spring 3.2 brings support for a global @ExceptionHandler with the @ControllerAdvice annotation

```
@ControllerAdvice
class BookAdvice {

  @ResponseBody
  @ExceptionHandler(BookException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  String bookHandler(BookException ex) {
   return ex.getMessage();
  }
}
```

```
public class BookException extends
RuntimeException {

  public BookException(String exception) {
   super(exception);
  }

}
```

Framework Springboot

Error Management

Same but with one class ?

```
@ControllerAdvice

public class RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {
@ExceptionHandler(value = { IllegalArgumentException.class, IllegalStateException.class })

protected ResponseEntity<Object> handleConflict( RuntimeException ex, WebRequest request) {
String bodyOfResponse = "This should be application specific";
return handleExceptionInternal(ex, bodyOfResponse, new HttpHeaders(),
HttpStatus.CONFLICT, request); } }
```

Source https://www.baeldung.com/exception-handling-for-rest-with-spring

Framework Springboot

Data Management with @Repository

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
</dependency>
```

Never write another getter or equals method again, with one annotation your class has a fully featured builder, Automate your logging variables, and much more.

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok  -->
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>xxxxAMODIFIERxxxxxx</version>
  <scope>provided</scope>
</dependency>
```

Framework Springboot

Data Management with @Repository

Spring Data repository abstraction takes the the domain class to manage as well as the id type of the domain class as type arguments.

CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed.

Framework Springboot

Data Management with @Repository

Entity

```
@Entity
@Data // -> pour lombok
public class Book {

        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        private  Long isbn;
@Column(name = "name")
        private String name;
@Column(name = "author")
        private String author;

Book() {
        this.name="";
        this.author="";

}
```

Framework Springboot

Data Management with @Repository

```
interface BookRepository extends CrudRepository<Book, Long> {

        List<Book> findByname(String name);
        Book findById(long id);
}
```

Use method save(Entity…) to add something

Framework Springboot

Data Management with @Repository

In the controller :

```
@RestController
public class BookController {

private final BookRepository repository;

BookController(BookRepository repository) {
   this.repository = repository;
                    repository.save(new Book("book", "Bauer"));
 }

@RequestMapping(value = "/books", method = RequestMethod.GET, produces = "application/json")
public @ResponseBody List<Book> getallbooks() {

List<Book> books = new ArrayList<>();
for(Book b: repository.findAll()) {
   books.add(b);
   }
return books;
}
```

Framework Springboot

Data Management with @Repository
By default (no BD) a MAP is created

To connect to BD
Add this to file application.properties

spring.datasource.url= jdbc:mysql://localhost:3306/testdb?useSSL=false
spring.datasource.username= root
spring.datasource.password= 123456

spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQL5InnoDBDialect

 # Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update

Springboot and DTO

DTO (Data Transfert Object)

* Design pattern « *An object that carries data between processes in order to reduce the number of method calls*»
* Gathers all the required data
* encapsulate the serialization mechanism

Springboot And DTO

DTO to Entity in @Service Service

```
@Service
public class UserService {
@Autowired
private BookRepository bookRepo;

public BookDTO getBookByIsbn(Long isbn) {
Book b = bookRepo.findByIsbn(isbn).orElse(null);
return convertToDTO(b);
}

public void saveBook(BookDTO bDTO) {
Book b = convertToEntity(bDTO);
bookRepo.save(b);
}

private UserDTO convertToDTO(Book b) {
// logic to convert User to BookDTO
}
etc.
}
```

Framework Springboot

Adding Hateos

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

Framework Springboot

Adding Hateos

Creates Links

Link link = Link.of("/something");

Springboot provides a builder WebMvcLinkBuilder to create links by pointing to controller classes

And get URI directly:

Person person = new Person(1L, "Dave", "Matthews"); //

Link link = linkTo(PersonController.class).slash(person.getId()).withSelfRel();

=> Génère le lien : localhost:8080/your-app/people/1

Framework Springboot

Adding Hateos

Another example :

```
@GetMapping("/books2/{isbn}")
 EntityModel<Book> one(@PathVariable Long isbn) {

   Book b = repository.findById(isbn) //
      .orElseThrow(() -> new BookException("not
found"));

   return EntityModel.of(b, //

linkTo(methodOn(BookController.class).one(isbn)).wi
thSelfRel(),

linkTo(methodOn(BookController.class).getallbooks()
).withRel("books"));
 }
```


```
isbn:       1
name:       "book"
author:     "Bauer"
▼ _links:
  ▼ self:
      href:   "http://localhost:8080/books2/1"
  ▼ books:
      href:   "http://localhost:8080/books"
```

Framework Springboot

Adding Log

Springboot includes Logback by default

Different levels:

```
@RestController public class LoggingController {
Logger logger = LoggerFactory.getLogger(LoggingController.class);
@RequestMapping("/") public String index(){
 logger.trace("A TRACE Message");
logger.debug("A DEBUG Message");
logger.info("An INFO Message"); logger.warn("A WARN Message");
 logger.error("An ERROR Message");
```

Cmd Options : -Dlogging.level.org.springframework=TRACE | DEBUG etc

Framework Springboot

Adding Log

Log management is very important (but not simple)

* Helps debugging
* Allows to compte metrix (performance; nb client requests ; energy -> green IT etc.)
* Gets data allowing to deduce how services and the company work ! (retro-engineering; data driven software engineeering; model learning ; etc.)

* Good logging Requires skills
  * How to log Web services ?
  * Which amount of data ?
  * Requires ids (every request/ response of the same transaction must include the same id, a.k.a. correlation set)

Framework Springboot

Adding Log

Springboot includes Logback by default

Permanently :

In the file application.properties :

logging.level.root=WARN

logging.level.com.baeldung=TRACE

In a file.xml in the resources directory

Framework Springboot

Adding Log
Springboot includes Logback by default

Permanently :

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

  <property name="LOGS" value="./logs" />

  <appender name="Console"
    class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>
        %black(%d{ISO8601}) %highlight(%-5level) [%blue(%t)] %yellow(%C{1}):
%msg%n%throwable
      </Pattern>
    </layout>
  </appender>

  <appender name="RollingFile"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${LOGS}/spring-boot-logger.log</file>
    <encoder
      class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
      <Pattern>%d %p %C{1} [%t] %m%n</Pattern>
    </encoder>

  <rollingPolicy
      class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <!-- rollover daily and when the file reaches 10 MegaBytes -->
      <fileNamePattern>${LOGS}/archived/spring-boot-logger-%d{yyyy-MM-dd}.%i.log
      </fileNamePattern>
      <timeBasedFileNamingAndTriggeringPolicy
        class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
        <maxFileSize>10MB</maxFileSize>
      </timeBasedFileNamingAndTriggeringPolicy>
    </rollingPolicy>
  </appender>

  <!-- LOG everything at INFO level -->
  <root level="info">
    <appender-ref ref="RollingFile" />
    <appender-ref ref="Console" />
  </root>

  <!-- LOG "" at TRACE level -->
  <logger name="example.serv" level="trace" additivity="false">
    <appender-ref ref="RollingFile" />
    <appender-ref ref="Console" />
  </logger>

</configuration>
```

Framework Springboot

Adding Log

Lib logbook : https://github.com/zalando/logbook

»enable complete request and response logging for different client- and server-side technologies.»

Allows to add filters on requests, responses, parameters to hide them

Springboot 2 at the moment

Returns logs in XML. Example :

{ "@timestamp": "2020-10-27T11:26:38.860+01:00", "@version": "1", "message": "{\"origin\":\"remote\",\"type\":\"request\",\"correlation\":\"b73a97811bd040f3\",\"protocol\":\"HTTP/1.1\",\"remote\":\"0:0:0:0:0:0:0:1\",\"method\":\"GET\",\"uri\":\"http://localhost:8080/companies?name=ippon\",\"headers\":{\"accept\":[\"/*\"],\"accept-encoding\":[\"gzip, deflate\"],\"cache-control\":[\"no-cache\"],\"connection\":[\"keep-alive\"],\"content-type\":[\"application/json\"]}}", "logger_name": "org.zalando.logbook.Logbook", "thread_name": "http-nio-8080-exec-2", "level": "TRACE", "level_value": 5000 }*

Framework Springboot

How to change the default port ?

Go to file ***application.properties and add*** server.port=8081

Springboot client side

Framework Springboot

Client generation :

Springboot provides a class to be used as a proxy

class RestTemplate:

- getForObject(url, classType)
- postForObject(url, request, classType)
- put(url, request)
- delete(url)

Example:

RestTemplate restTemplate = new RestTemplate();

String result = restTemplate.getForObject(uri, String.class); // Object returned by Web service : here a string

Framework Springboot

Complete example :

```
@SpringBootApplication
public class DemoApplication{
        private static final Logger log = LoggerFactory.getLogger(DemoApplication.class);

        public static void main(String[] args) {

RestTemplate restTemplate = new RestTemplate();
Book b=new Book("3","toto","toto");
restTemplate.postForObject("http://localhost:8080/postbook",b , Book.class);

ArrayList<Book> l = restTemplate.getForObject(
                "http://localhost:8080/getallbooks", ArrayList.class);
                System.out.println (l.toString());

log.info(l.toString());
}}
```

Framework Springboot

Security

Spring Security

basicAuth (login, passworg in header)

https://spring.io/guides/gs/securing-web/

Protocol Oauth v2

In a nuth-shell:

Login;secret + generation of token

Requires :

* a table in DB. Why ?
* A lot of configuration files
* Authorisation on methods : @PreAuthorize("#oauth2.hasScope('read')")

Microservices (micronaut)

Micronaut

* https://micronaut.io/

* Micro- Framework
  * memory consumption
  * Built-in cloud support
  * use GraalVM
  * Generates docker config files
  * In early stages

* Provide a client *mn* to generate projects

Micronaut

* Is also based upon a root app and controllers
* Root app:

```
public class Application {
   public static void main(String[] args) {
      Micronaut.run(Application.class, args);
   }
}
```

Micronaut

* Exemple Service. Annotations very similar to Springboot?

```
@Controller("/cont")
public class ContController {

   @Get(uri="/hello", produces="text/plain")
   public String index() {
      return "Example Response";
   }

  @Get(uri="/savegetEntity/{texte}", produces="text/html")
public String savegetEntity(@PathVariable String texte) {
…]
```

Micronaut

* Error management
* Build HTTP responses

return HttpResponse.status(HttpStatus.UNAUTHORIZED).body(m);

Or your own exceptions

Micronaut

Or your own exceptions

```
package example;

public class ExampleException extends
RuntimeException {
}
```

```
@Produces
@Singleton
@Requires(classes = {ExampleException.class,
ExceptionHandler.class})
public class ExampleExceptionHandler
implements
ExceptionHandler<ExampleException,
HttpResponse> {

  @Override
  public HttpResponse handle(HttpRequest
request, ExampleException exception) {
    return
HttpResponse.status(HttpStatus.INTERNAL_SER
VER_ERROR, "erreur gravissime");
  }
}
```

Micronaut

Scopes

@Singleton	Singleton scope indicates only one instance of the bean should exist
@Context	Context scope indicates that the bean should be created at the same time as the ApplicationContext (eager initialization)
@Refreshable	@Refreshable scope is a custom scope that allows a bean's state to be refreshed via the /refresh endpoint.
@RequestScope	@RequestScope scope is a custom scope that indicates a new instance of the bean is created and associated with each HTTP request

Micronaut

* *Code a client ?*
* *Well… yes but inside a controller (?)*

```
@Controller("/cl")
public class HelloController {

  private MicrotestClient testClient;

  public HelloController(MicrotestClient testClient) {
    this.testClient = testClient;}
  @Get("/")
  public String index() {
   try{
    return testClient.hello();
}catch (HttpClientResponseException e){return "error: "+ e.getMessage();}
  }
}
```

Micronaut

* Code a client ?
* Well… yes but inside a controller (?)

* Interface client example :

```
@Client("http://localhost:8080/cont/hello")
public interface MicrotestClient {

  @Get(consumes = MediaType.TEXT_PLAIN)
  String hello();
```

Testing

* Selenium
* Postman

* https://rest-assured.io/
* https://github.com/intuit/karate
* https://citrusframework.org/ (integration testing, BD)

Performance
* http://jmeter.apache.org/index.html
* https://gettaurus.org/

Mock
* https://github.com/mock-server/mockserver
* https://hoverfly.io/