Spring Boot

UA.DETI.IES - 2019/20

Main topics

- Spring Boot

- Spring Boot dependencies, auto-configuration and runtime
- Spring MVC Architecture

- Spring Data
- Spring Data JPA
- JDBC, Hibernate, MongoDB

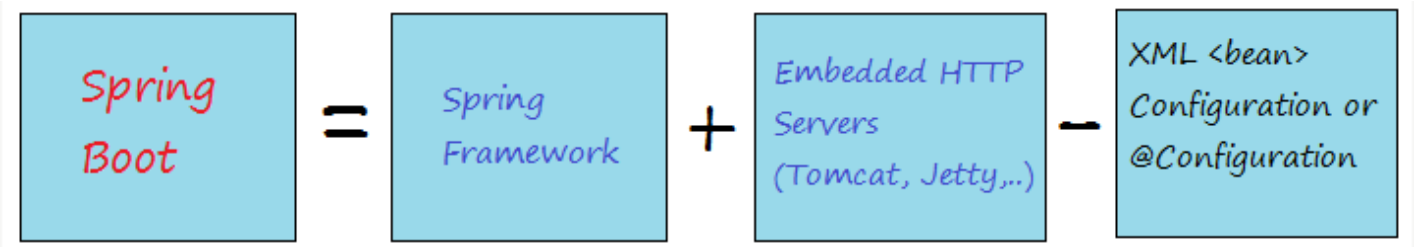
What is Spring?

- Simply put, the Spring framework provides comprehensive infrastructure support for developing Java applications.
- It is packed with some nice features like Dependency Injection and out of the box modules like:
  - Spring JDBC
  - Spring MVC
  - Spring Security
  - Spring AOP
  - Spring ORM
  - Spring Test
- These modules can reduce the development time of an application.
  - For example, in the early days of Java web development, we needed to write a lot of code to insert a record into a data source.
  - But by using the JDBCTemplate of the Spring JDBC module we can reduce it to a few lines of code with only a few configurations.

https://www.baeldung.com/spring-vs-spring-boot

What is Spring Boot?

- Extension of the Spring framework
  - that eliminated (even more) the boilerplate configurations required for setting up a Spring application.
  - It takes an opinionated view of the Spring platform, for a faster and more efficient development eco-system.
- Some features:
  - Opinionated 'starter' dependencies to simplify build and application configuration
  - Embedded server to avoid complexity in application deployment
  - Automatic config for Spring functionality

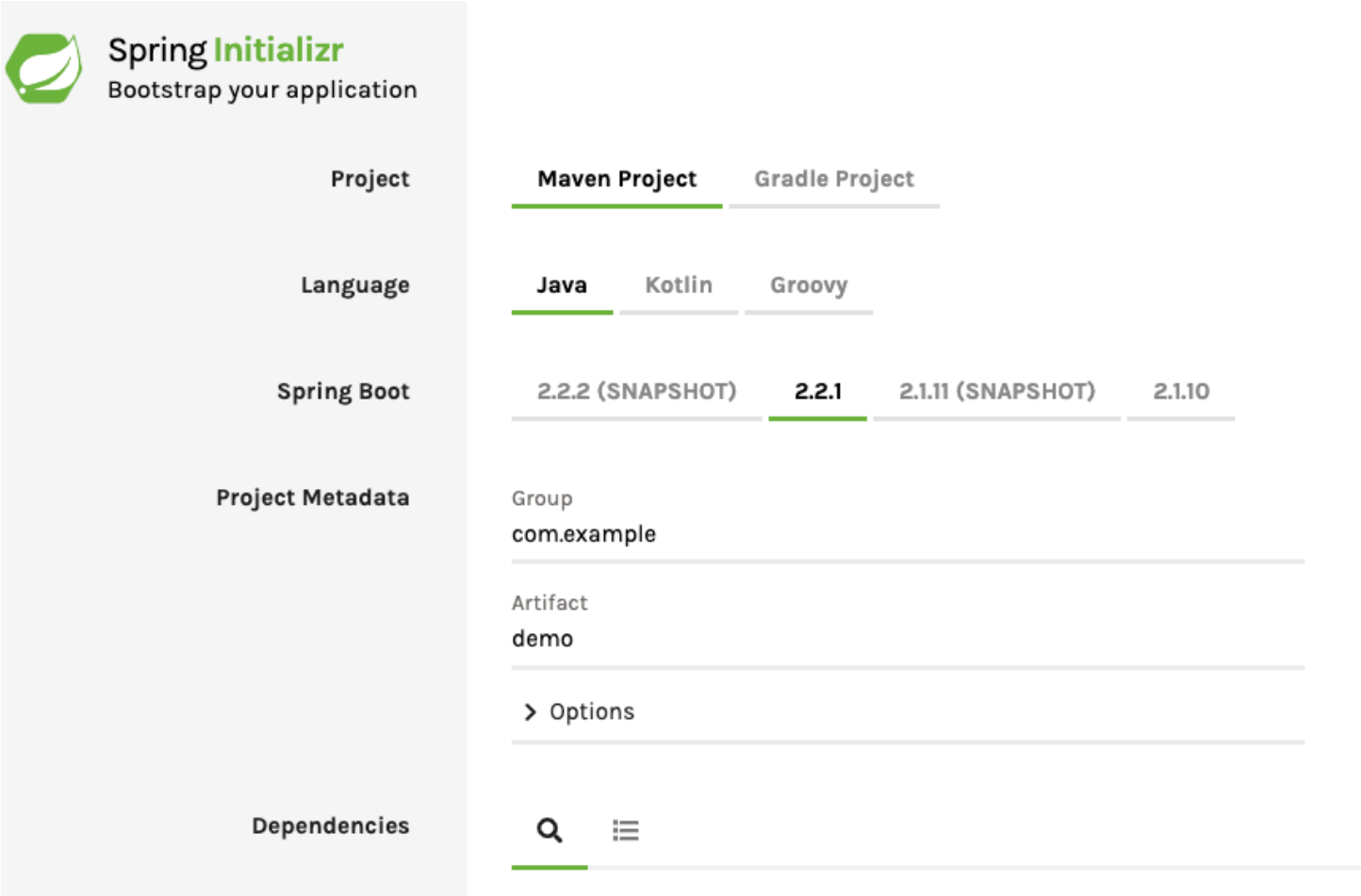


Spring Boot Main Components

- **Starters** - combine a group of common or related dependencies into single dependency
- **AutoConfigurator** - reduce the Spring Configuration
- **CLI** - run and test Spring Boot applications from command prompt
- **Actuator** – provides EndPoints and Metrics
  - E.g. http://localhost:8080/actuator/health
    - {"status":"UP"}

Creating a Spring Boot project

- Spring Initializr
  - https://start.spring.io



Spring Boot main application

```
@SpringBootApplication
public class PayrollApplication {

    public static void main(String... args) {
        ApplicationContext ctx =
          SpringApplication.run(PayrollApplication.class, args);
    }
}
```

Enable component-scanning and auto-configuration

Bootstrap the application

@SpringBootApplication

- Enables Spring component-scanning and Spring Boot auto-configuration, by combining three other annotations:
- @Configuration
  - Designates a class as a configuration class using Spring's Java-based configuration.
- @ComponentScan
  - Enables component-scanning so that the web controller classes and other components will be automatically discovered and registered as beans in the Spring application context.
- @EnableAutoConfiguration
  - It enables the "magic" of Spring Boot auto-configuration avoiding to write the pages of XML configuration that would be required otherwise.

Starters

- Starters are a set of pre-defined dependency descriptors
  - They avoid having to copy-paste loads of dependencies.
- All official starters follow a similar naming pattern
  - *spring-boot-starter-**, where * is a particular type of application.
- Examples
  - spring-boot-starter (core starter)
  - spring-boot-starter-web (Spring MVC)
  - spring-boot-starter-amqp (RabbitMQ)
  - spring-boot-starter-data-jpa (JPA, hibernate)

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter

Starters

- spring-boot-starter-jdbc
  - Traditional JDBC applications
- spring-boot-starter-hateoas
  - Make your services more RESTful by adding HATEOAS features
- spring-boot-starter-web-services
  - For building applications exposing SOAP web services
- spring-boot-starter-test
  - Write great unit and integration tests
- spring-boot-starter-security
  - Authentication and authorization using Spring Security
- spring-boot-starter-cache
  - Enabling the Spring Framework's caching support
- spring-boot-starter-data-rest
  - Expose simple REST services using Spring Data REST

POM.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       https://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>ies.spring</groupId>
   <artifactId>demo</artifactId>
   <version>0.0.1-SNAPSHOT</version>

   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.1.9.RELEASE</version>
      <relativePath /> <!-- lookup parent from repository -->
   </parent>
```

Inherit versions from starter parent

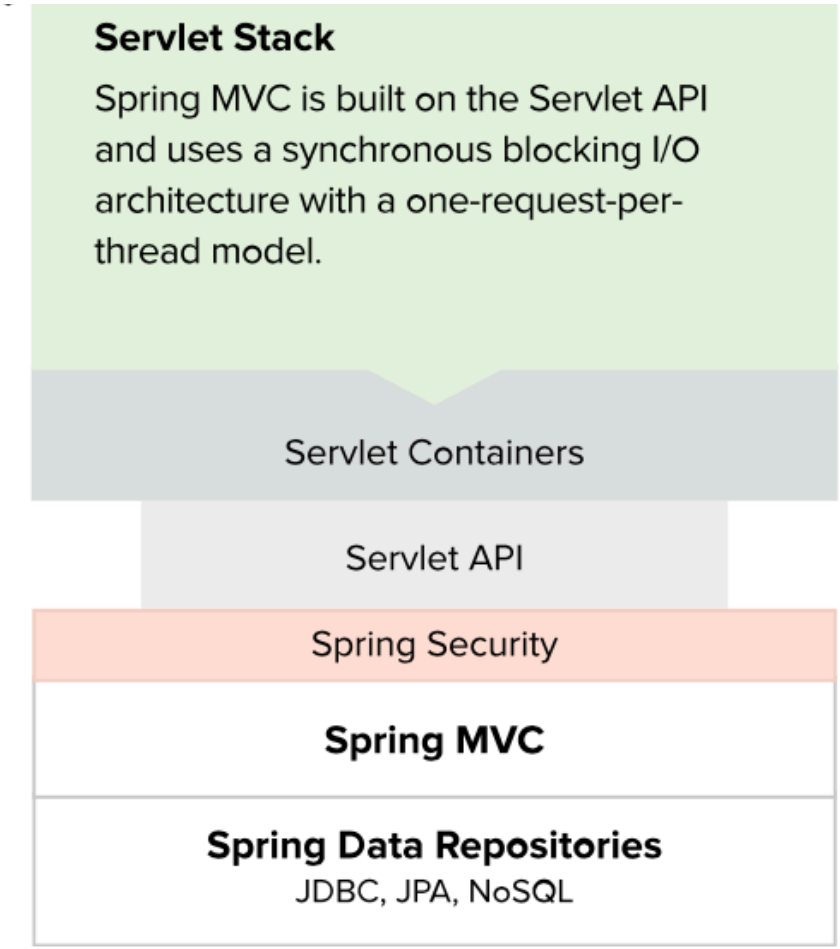
POM.xml - dependencies

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.2</version>
            <scope>provided</scope>
        </dependency>

    ...
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

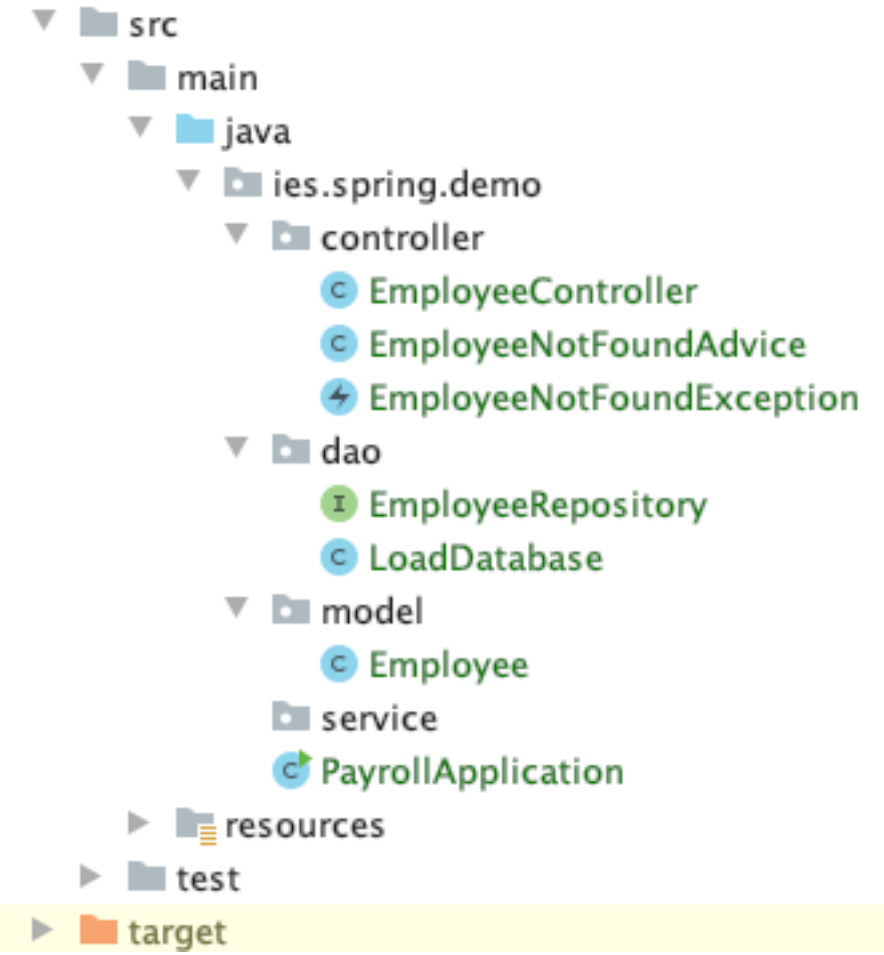
The Spring Web MVC

- Spring MVC allows creating special @Controller or @RestController beans to handle incoming HTTP requests.
- Methods in the controller are mapped to HTTP by using @RequestMapping annotations.



The Spring Web MVC

- Project structure - example



model.Employee.java

```java
package ies.spring.demo.model;

import lombok.Data;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Data
@Entity
public class Employee {

  private @Id @GeneratedValue Long id;
  private String name;
  private String role;

  Employee() {}

  public Employee(String name, String role) {
    this.name = name;
    this.role = role;
  }
}
```

@Entity denotes this is an entity object for the table name Employee

The field id is the Primary Key and, hence, marked as **@Id**.

The field id is also marked with **@GeneratedValue**, which denotes that this is an Auto-Increment column.

dao.EmployeeRepository

```
package ies.spring.demo.dao;

import ies.spring.demo.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository
      extends JpaRepository<Employee, Long> {
         List<Employee> findByName(String name);
}
```

Where is this methods implemented?

- The *JpaRepository* interface is parameterized with two parameters:
  - the domain type that the repository will work with, and the type of its ID property.
- *EmployeeRepository* inherits 18 methods for performing common persistence operations.
  - In addition, we may add other methods.
- The interface will be implemented automatically by Spring Boot at runtime when the application is started.

Derived Query Methods

- Derived method names have two main parts separated by the first *By* keyword:
  - The first part – like *find* – is the introducer and the rest – like *ByName* – is the criteria.

```
List<Employee> findByName(String name)
```

- Spring Data JPA supports *find*, *read*, *query*, *count* and *get*.
  - we could have done *queryByName* and Spring Data would behave the same.
- We can also use *Distinct*, *First*, or *Top* to remove duplicates or limit our result set:

```
List<Employee> findTop3ByAge()
```

Query methods: some examples

- findByLastnameAndFirstname
- findByLastnameOrFirstname
- findByStartDateBetween
- findByAgeLessThan
- findByStartDateAfter
- findByStartDateBefore
- findByAgeIsNull
- findByFirstnameLike
- findByFirstnameStartingWith
- findByAgeOrderByLastnameDesc
- findByAgeIn(Collection<Age> ages)
- findByFirstnameIgnoreCase

Controller.EmployeeController

```
@RestController
public class EmployeeController {
  private final EmployeeRepository repository;

  EmployeeController(EmployeeRepository repository) {
    this.repository = repository;
  }
  @GetMapping(value = "/all", produces = "application/json; charset=UTF-8")
  List<Employee> allMembers() {
    return repository.findAll();
  }
  @GetMapping(value = "/employees")
  Resources<Resource<Employee>> employees() {
    List<Resource<Employee>> employees = repository.findAll().stream()
      .map(employee -> new Resource<>(employee,
                linkTo(methodOn(EmployeeController.class)
      .one(employee.getId())).withSelfRel(),
                linkTo(methodOn(EmployeeController.class)
      .all()).withRel("employees")))
      .collect(Collectors.toList());

    return new Resources<>(employees,
      linkTo(methodOn(EmployeeController.class).all()).withSelfRel());
  }
```

Annotation Type GetMapping

```
@GetMapping(value = "/all", produces = "application/json;
            charset=UTF-8")
List<Employee> allMembers() {
    return repository.findAll();
}
```

- org.springframework.web.bind.annotation

```
@Target(value=METHOD)
@Retention(value=RUNTIME)
@Documented
@RequestMapping(method=GET)
public @interface GetMapping {
   String[] consumes
   String[] headers
   String   name
   String[] params
   String[] path
   String[] produces
   String[] value
}
```

@GetMapping is a composed annotation,
i.e. a shortcut for
@RequestMapping(method = RequestMethod.GET).

AutoConfigurator

- Example, to create an application that accesses a relational database with JDBC:

```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
}
```

  - This declaration creates an instance of JdbcTemplate, injecting it with its one dependency, a DataSource.
  - That means that you'll also need to configure a DataSource bean so that the dependency will be met.
- To complete this configuration scenario, we may use an embedded H2 database as the DataSource bean:

```
@Bean
public DataSource dataSource() {
   return new EmbeddedDatabaseBuilder()
   .setType(EmbeddedDatabaseType.H2).addScripts('schema.sql', 'data.sql').build();
}
```

  - This bean configuration method creates an embedded database, specifying two SQL scripts to execute on the embedded database.

Running the application

- One of the biggest advantages of packaging an application as a jar and using an embedded HTTP server is that we can run it as:

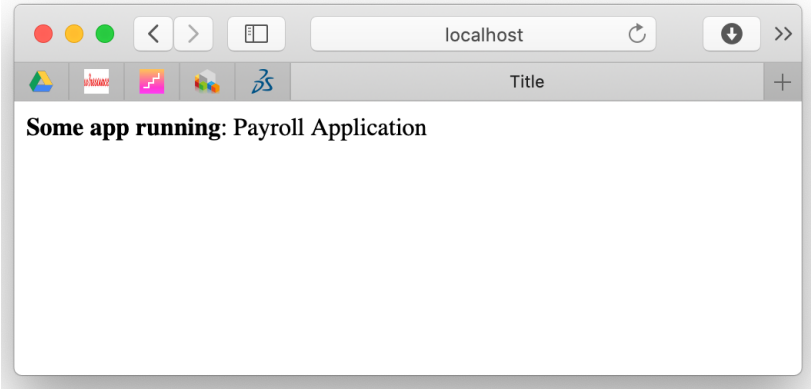
```
$ java -jar target/demo-0.0.1-SNAPSHOT.jar
```

```
.   ____          _            __ _ _
/\\ / ___'_ __ _()_ __  __ _ \ \ \ \
( ( )\___ | ' | '_| | ' \/ _` | \ \ \ \
\\/  ___)| |_)| | | | | || (_| |  ) ) ) )
 '  |____| .__|_| |_|_| |_\__, | / / / /
=========|_|==============|___/=///_/
:: Spring Boot ::        (v2.1.9.RELEASE)
2019-11-11 14:37:45.322  INFO 20408 --- [           main]
ies.spring.demo.PayrollApplication       : Starting PayrollApplication v0.0.1-SNAPSHOT on
staff-0118.wireless.ua.pt with PID 20408 (
...
2019-11-11 14:37:45.454  INFO 21552 --- [           main]
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with
context path ''
...
```

Using the app

```
$ curl -v localhost:8080/employees

*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> GET /employees HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
< Transfer-Encoding: chunked
< Date: Mon, 11 Nov 2019 17:15:19 GMT
<
* Connection #0 to host localhost left intact
{"_embedded":{"employeeList":[{"id":1,"name":"Bilbo
Baggins","role":"burglar","_links":{"self":{"href":"http://localhost:8080/employees
/1"},"employees":{"href":"http://localhost:8080/employees"}}},{"id":2,"name":"Frodo
Baggins","role":"thief","_links":{"self":{"href":"http://localhost:8080/employees/2
"},"employees":{"href":"http://localhost:8080/employees"}}}]},"_links":{"self":{"hr
ef":"http://localhost:8080/employees"}}}* Closing connection 0
```



Using the app

```
$ curl -v localhost:8080/employees/99
   Trying ::1...
   ...
   Could not find employee 99* Closing connection 0

$ curl -X POST localhost:8080/employees -H 'Content-type:application/json'
-d '{"name": "Samwise Gamgee", "role": "gardener"}'
   {"id":3,"name":"Samwise Gamgee","role":"gardener"}

$ curl -X GET localhost:8080/employees/3
   {"id":3,"name":"Samwise
   Gamgee","role":"gardener","_links":{"self":{"href":"http://localhost:808
   0/employees/3"},"employees":{"href":"http://localhost:8080/employees"}}}
   * Closing connection 0

$ curl -X PUT localhost:8080/employees/3 -H 'Content-type:application/json'
-d '{"name": "Samwise Gamgee", "role": "ring bearer"}'
   {"id":3,"name":"Samwise Gamgee","role":"ring bearer"}
```

Spring WebFlux Framework

- Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0.
  - It is fully asynchronous and non-blocking, and implements the Reactive Streams specification (Reactor lib).
- Essentially, reactive streams is a specification for asynchronous stream processing.
  - In other words, a system where lots of events are being produced and consumed asynchronously.
- Spring WebFlux comes in two flavors: functional and annotation-based.
  - The annotation-based one is quite close to the Spring MVC model, as shown in the following example.

Spring WebFlux Framework

```
@RestController
@RequestMapping("/users")
public class MyRestController {

    @GetMapping("/{user}")
    public Mono<User> getUser(@PathVariable Long user) {
        // ...
    }

    @GetMapping("/{user}/customers")
    public Flux<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping("/{user}")
    public Mono<User> deleteUser(@PathVariable Long user) {
        // ...
    }

}
```

Mono<T> emits 0..1 elements

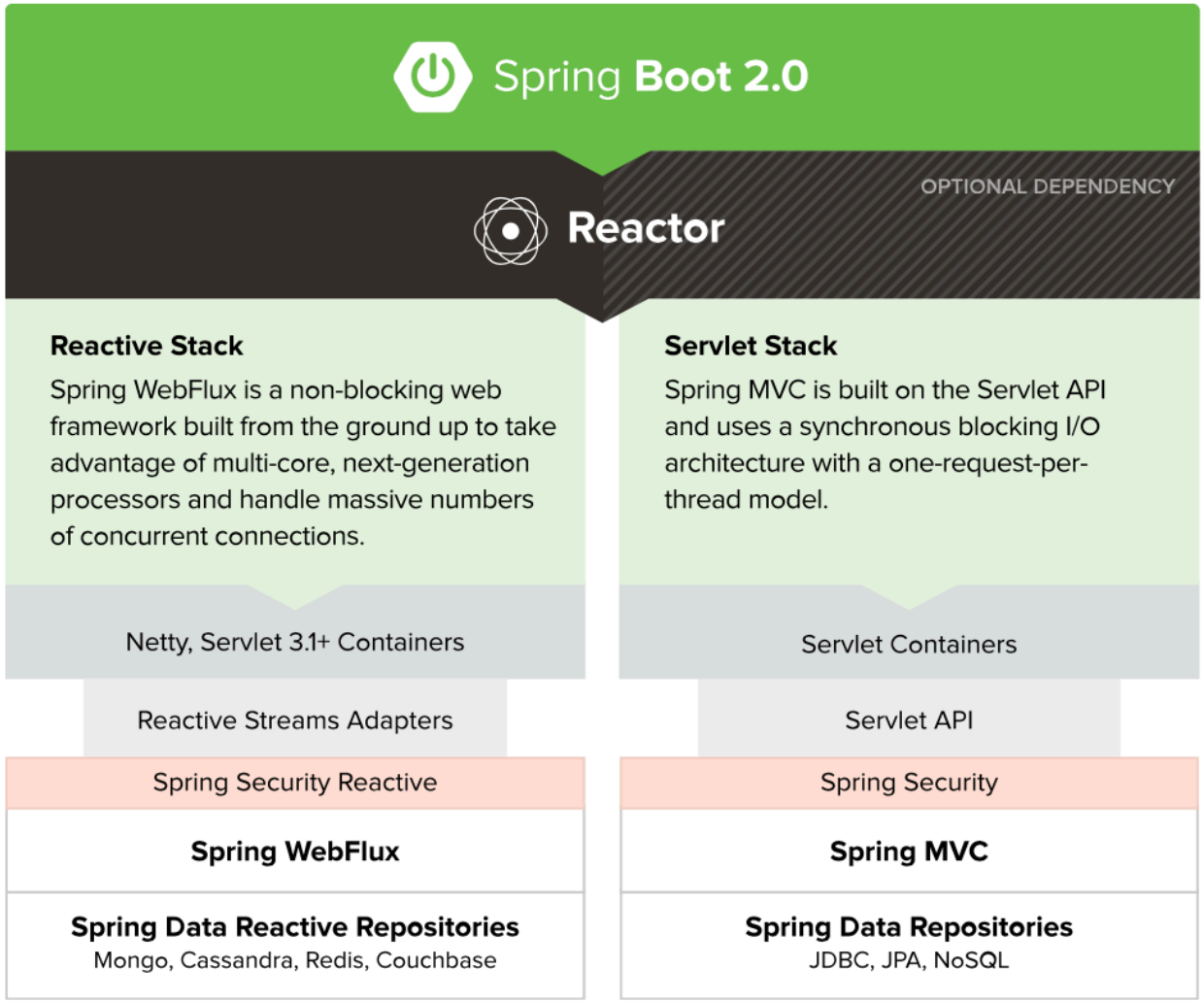
Flux<T> emits 0..n elements

WebFlux - Functional variant

```
@Configuration(proxyBeanMethods = false)
public class RoutingConfiguration {
   @Bean
   public RouterFunction<ServerResponse> monoRouterFunction(UserHandler userHandler) {
     return route(GET("/{user}").and(accept(APPLICATION_JSON)), userHandler::getUser)
       .andRoute(GET("/{user}/customers")
             .and(accept(APPLICATION_JSON)), userHandler::getUserCustomers)
       .andRoute(DELETE("/{user}")
             .and(accept(APPLICATION_JSON)), userHandler::deleteUser);
    }
}

@Component
public class UserHandler {
    public Mono<ServerResponse> getUser(ServerRequest request) {
        // ...
    }
    public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
        // ...
    }
    public Mono<ServerResponse> deleteUser(ServerRequest request) {
        // ...
    }
}
```

Spring MVC vs. WebFlux



Spring Data

Persistence

- **Spring ORM**
- The *ORM* package is related to the database access. It provides integration layers for popular object-relational mapping APIs (e.g. JDO, Hibernate).
- **Spring DAO**
- The DAO (Data Access Object) support in Spring is primarily for standardizing the data access work using the technologies like JDBC, Hibernate or JDO.

Persistence

- Java Persistence API (JPA)
- Spring Data JPA
- Hibernate Framework
- JDBC
- MongoDB
- Transactions

Persistence

The persistence layer is vital to almost any application. Have a look at different ways to implement persistence in Java.

JPA (63) | Hibernate (58) | MongoDB (17) | Couchbase (9) | JDBC (8) | Transactions (8)

https://www.baeldung.com/category/persistence/

Java Persistence API (JPA)

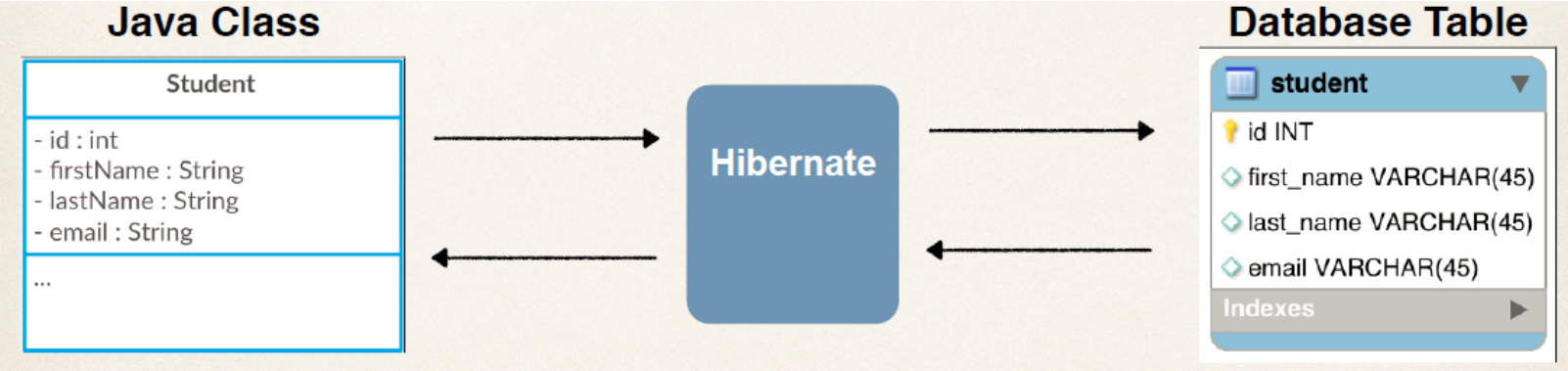
- The Java Persistence API (JPA) is the Java standard for mapping Java objects to a relational database.
  - It includes specifications, the entity and association mappings, the entity lifecycle management, and JPA's query capabilities
  - Mapping Java objects to database tables and vice versa is called Object-relational mapping (ORM).
- The Java Persistence API (JPA) is one possible approach to ORM.
  - Via JPA the developer can map, store, update and retrieve data from relational databases to Java objects and vice versa.
- Popular implementations are Hibernate, EclipseLink and Apache OpenJPA.

Spring Data JPA

- Spring Data JPA is not an implementation of JPA
  - It adds another layer on top of JPA.
  - It adds its own features like a no-code implementation of the repository pattern and the creation of database queries from method names.
  - It is still an abstraction used to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.
- It offers a solution to *GenericDao* custom implementations.
- It can also generate JPA queries on your behalf through method name conventions.

Hibernate

- Hibernate is a Java-based ORM tool
  - provides a framework for mapping application domain objects to the relational database tables and vice versa.
- Hibernate provides a reference implementation of the Java Persistence API that makes it a great choice as ORM tool with benefits of loose coupling.



JPA with Hibernate

- Spring Boot configures Hibernate as the default JPA provider
  - To enable JPA in a Spring Boot application, we need the s***pring-boot-starte***r and ***spring-boot-starter-data-jpa*** dependencies:
- Spring Boot can also auto-configure the ***dataSource*** bean, depending on the database we're using.
  - For in-memory database (e.g. H2), Boot automatically configures the *dataSource*.
  - we only need to add the H2 dependency to the pom.xml file.

@Entity

- Entities in JPA are nothing but POJOs representing data that can be persisted to the database.
  - Assuming we have:

```
public class Customer {
    private Long id;
    private String username;
    private String password;
    private String full_name;
    private Integer age;
    ...
}
```

  - We must ensure that the entity has a no-arg constructor and a primary key
  - Entity classes must not be declared final.
- An entity represents a table stored in a database.
  - Every instance of an entity represents a row in the table.

@Entity

```
@Entity(name="customer")
@Table(name = "CUSTOMERS", schema = "CHAINS") // namespace
public class Customer {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private Long id;

   private String username;
   @Column (nullable = false)
   private String password;
   @Column (name = "name", length=50, nullable = false)
   private String full_name;
   @Transient
   private Integer age;  // not persistent (or static, final, transient)
     ...
}
```

Embeddable Classes

- Embeddable classes are user defined classes that function as value types.
  - As with other non entity types, instances of an embeddable class can only be stored in the database as embedded objects, i.e. as part of a containing entity object.
- A class is declared as embeddable by marking it with the Embeddable annotation:

```
@Embeddable
public class Address {
    String street;
    String city;
    String state;
    String country;
    String zip;
}
```

Relationships

- Every persistent field can be marked with one of the following annotations:

- @OneToOne, @ManyToOne
  - for references of entity types.
- @OneToMany, @ManyToMany
  - for collections and maps of entity types.
- @MappedSuperClass / @Inheritance
  - for inherited types

@OneToMany / @ManyToOne

- The following entity classes demonstrate a bidirectional relationship:


```
@Entity
class Customer {
    ...
    @OneToMany (mappedBy = "customer")  // owned
    Set<Order> orders;
}

@Entity
class Order {
    ...
    @ManyToOne (optional = false)  // owning
    Customer customer;
    ...
}
```

Owned entity always maps to owning entity!

@ManyToMany


```
@Entity
class Product {
    @Id
    @GeneratedValue (strategy = GenerationType.SEQUENCE)
    @Column (name = "pid")
    long prod_id;

    @Column (nullable = false)
    float price;

    @ManyToMany (mappedBy = "products",  // owned
                 fetch = FetchType.EAGER)
    Set<Order> orders;
}

@Entity
class Order {
    ...
    @ManyToMany (fetch = FetchType.EAGER)  // owning
    @JoinTable (name = "OrderLines",
                joinColumns = @JoinColumn (name = "oid"),
                inverseJoinColumns = @JoinColumn (name = "pid"))
    Set<Product> products;
    ...
}
```

Spring Data JPA @Query

- We can use the @Query annotation to execute both JPQL and native SQL queries
  - SQL native – *over JDBC*

```
@Query( value = "SELECT * FROM USERS u WHERE u.status = 1",
        nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

  - JPQL (*JPA Query Language*) – *over Hibernate*

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();

@Query(value = "SELECT u FROM User u")
List<User> findAllUsers(Sort sort);

@Query(value = "SELECT u FROM User u ORDER BY id")
Page<User> findAllUsersWithPagination(Pageable pageable);
```

Sorting

Pagination

https://www.baeldung.com/spring-data-jpa-query

Programmatic querying

- **JPA Criteria API** – example: "*select * from Item*"

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Item> cr = cb.createQuery(Item.class);
Root<Item> root = cr.from(Item.class);
cr.select(root);
Query<Item> query = session.createQuery(cr);
List<Item> results = query.getResultList();
```

- **Querydsl** - example
  - For each @Entity, e.g. Person, Querydsl generates a query type with the simple name QPerson into the same package as Person.

```
QPerson person = QPerson.person;
List<Person> persons =
  query.from(person).where(person.firstName.eq("Kent")).list(person);
```

JPA with MySQL database

- We need the ***mysql-connector-java*** dependency, as well as to define the ***DataSource*** configuration.
  - We can do this in a @Configuration class:

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
    dataSource.setUsername("mysqluser");
    dataSource.setPassword("mysqlpass");
    dataSource.setUrl(
      "jdbc:mysql://localhost:3306/myDb?createDatabaseIfNotExist=true");
    return dataSource;
}
```

  - or by using a properties file, prefixed with *spring.datasource*:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=mysqluser
spring.datasource.password=mysqlpass
spring.datasource.url=
jdbc:mysql://localhost:3306/myDb?createDatabaseIfNotExist=true
```

https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa

JPA with MongoDB

- Define the model (as previously)
  - e.g. Person

```
@Document(collections = "school")
public class Person {
  @Id
  private ObjectId id;
  private Integer ssn;
  @Indexed
  private String name;
}
```

we can't use @GeneratedValue annotation, as it's not available.

- Adding Repository

```
@Repository
public interface PersonRepository
     extends MongoRepository<Person, String> {
        Person findByName(String name);
}
```

JPA with MongoDB

- Adding connection info in application.properties

```
spring.data.mongodb.host=[host]
spring.data.mongodb.port=[port]
spring.data.mongodb.authentication-database=[authentication_database]
spring.data.mongodb.username=[username]
spring.data.mongodb.password=[password]
spring.data.mongodb.database=some_database
```

- Create the REST Controller

- Querying

```
@Query("{'name : ?0'}")
Employee findByName(String name);
```

Spring Data - summary

- Spring Data consists of many independent projects, e.g.:
  - Spring Data Commons
  - Spring Data JPA
  - Spring Data KeyValue
  - Spring Data LDAP
  - Spring Data MongoDB
  - Spring Data Redis
  - Spring Data REST
  - Spring Data for Apache Cassandra
  - Spring Data for Apache Solr
  - Spring Data Couchbase (community module)
  - Spring Data Elasticsearch (community module)
  - Spring Data Neo4j (community module)

References

- https://spring.io/projects/spring-boot
- https://spring.io/projects/spring-data

- https://www.baeldung.com/spring-tutorial
- https://www.baeldung.com/persistence-with-spring-series

- https://www.edureka.co/blog/spring-tutorial/

- *... and many others*