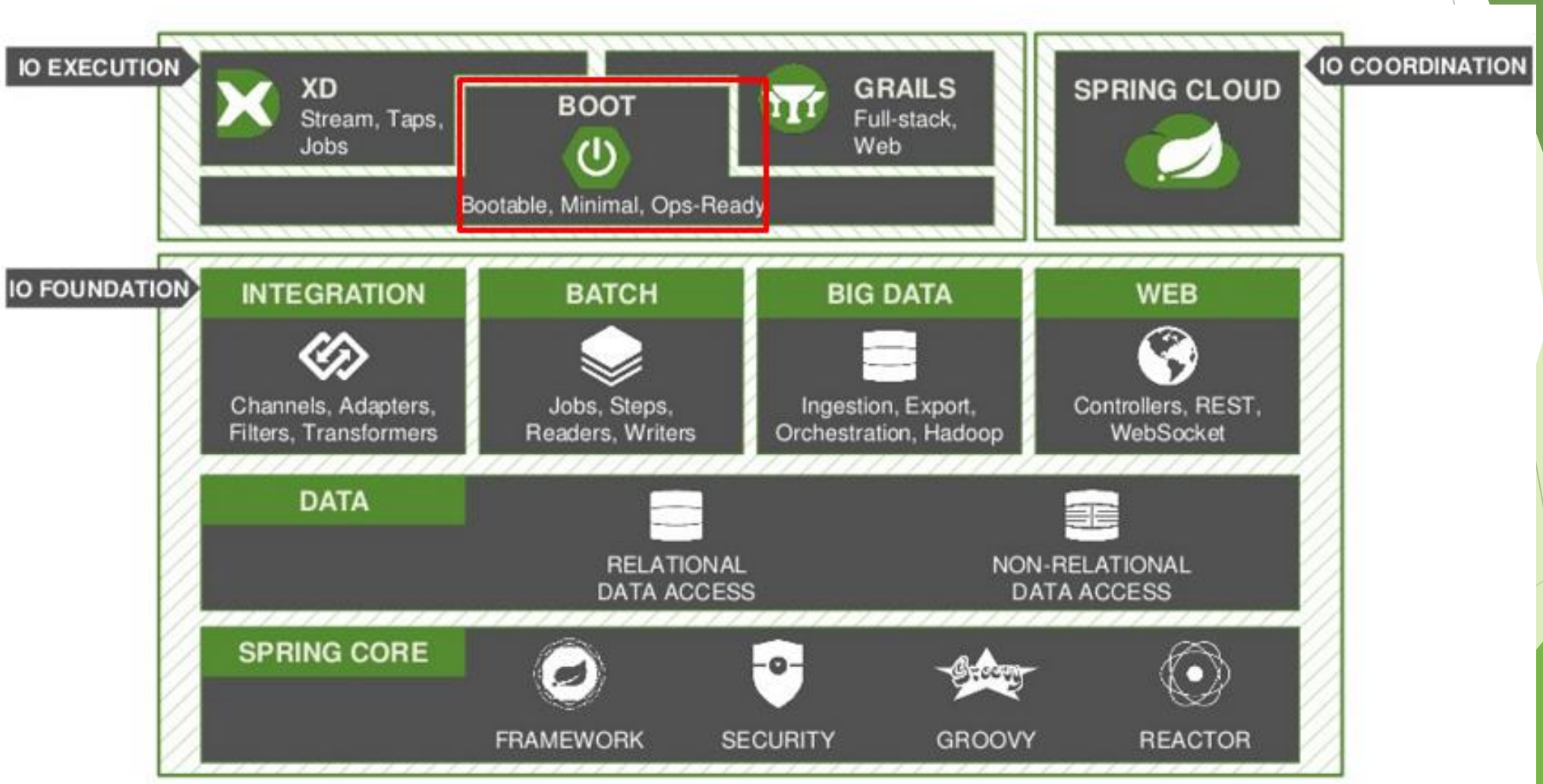
Spring Web

Spring Boot

Spring MVC

Spring Data JPA

Що таке Spring Boot?



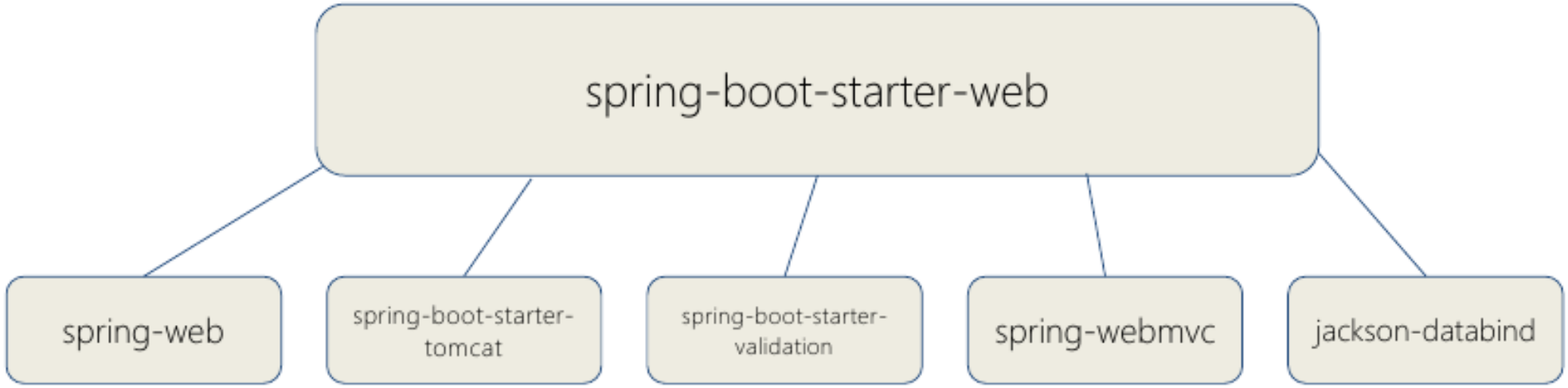
Spring Boot Application

- Використовує сценарії збірки Maven або Graddle.
- Об’єднує залежності в зручні стартер-пакети для окремих задач: web, jdbc, jpa, і т.п.
- Головний клас @SpringBootApplication запускає контекст Spring, скануючи при цьому компоненти з класів конфігурації власного пакета.
- Поверх контексту Spring Boot можна запускати Java Web Server Tomcat (або Jetty)
- Структуру проекту можна задати в Intellij Idea Ultimate, або на сайті start.spring.io і розробляти з використанням Intellij Idea Community, чи іншої IDE

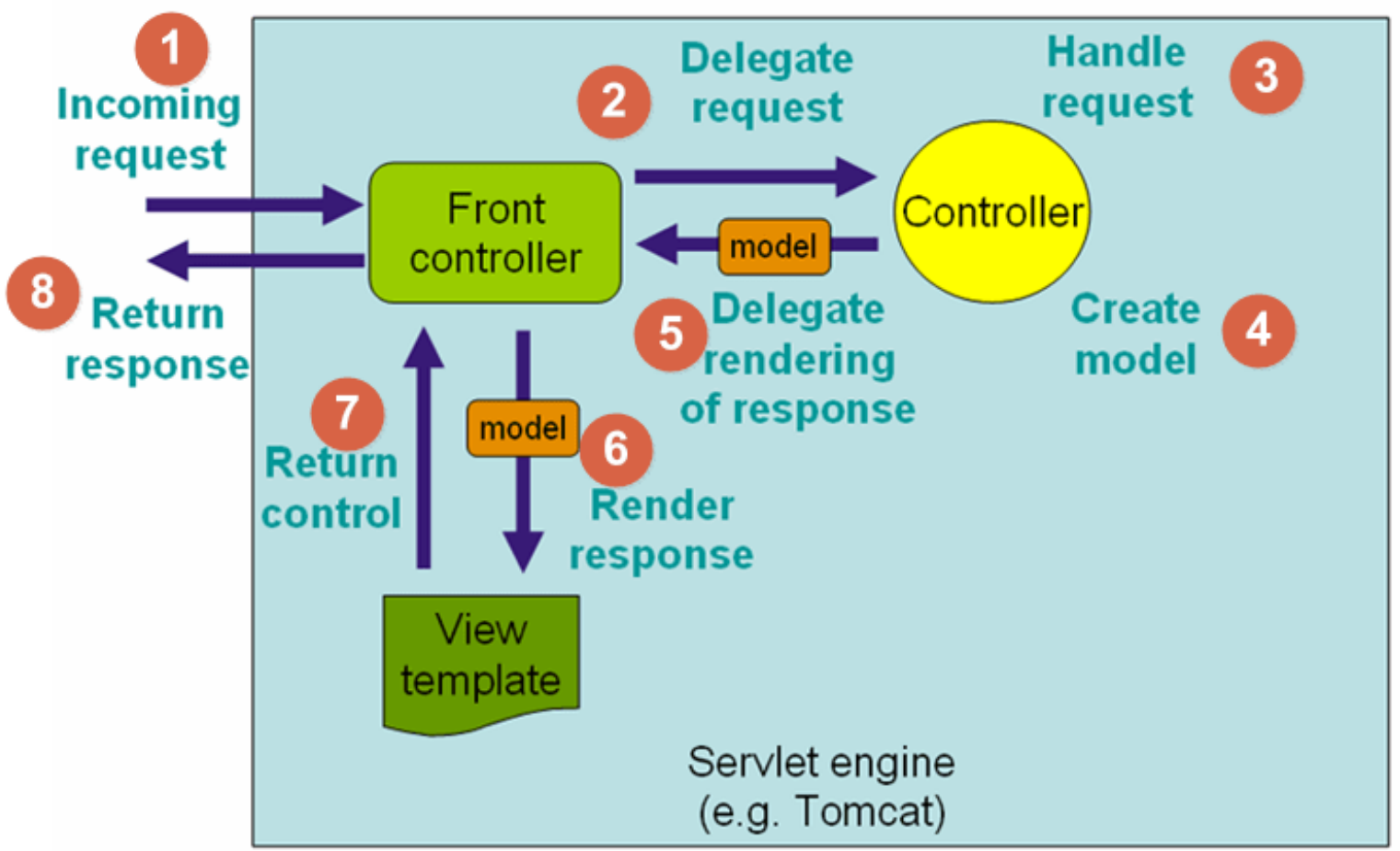
Стартер-пакети

Spring MVC

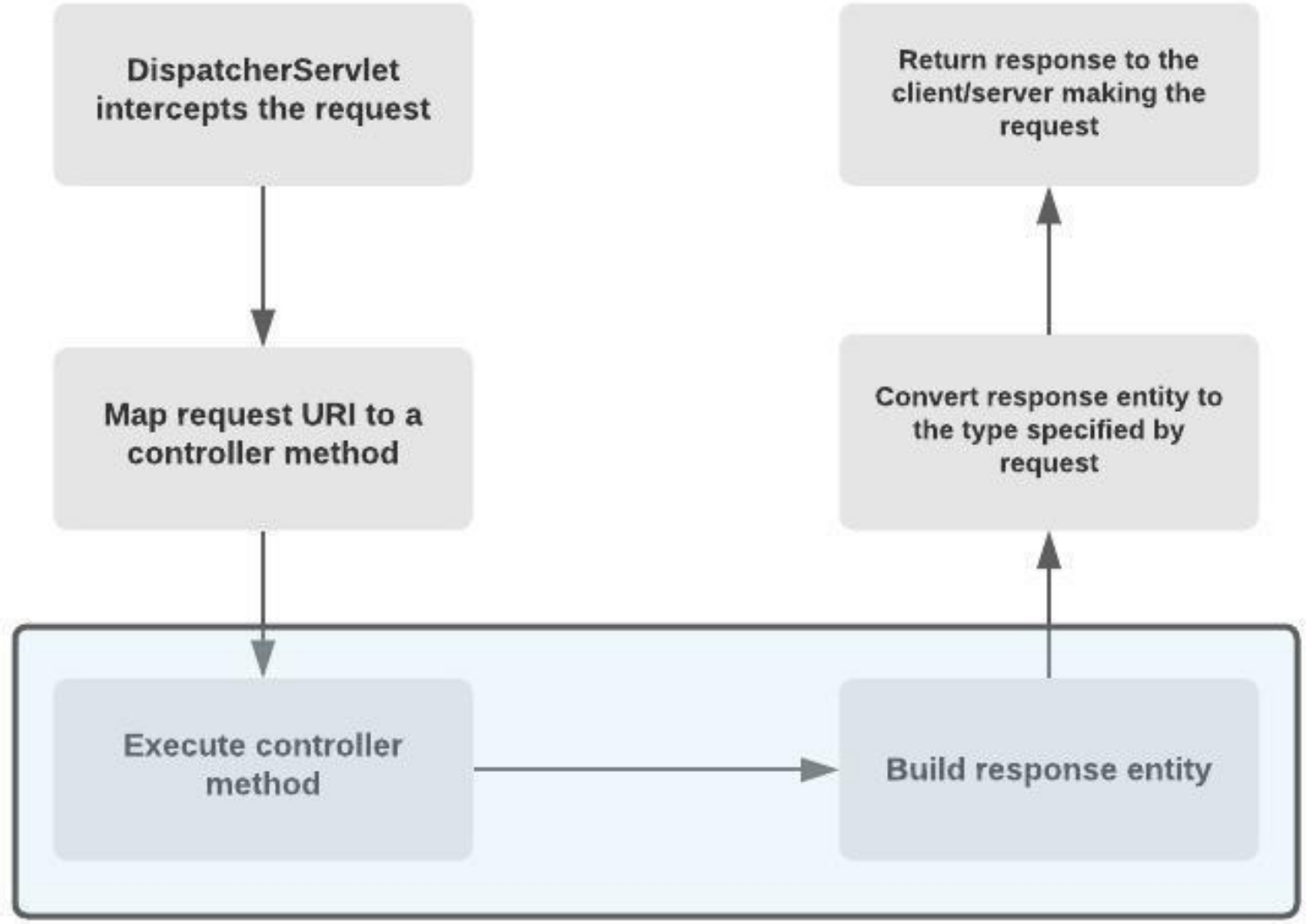
Spring Boot пакети



Spring MVC


1
Incoming request
2
Delegate request
Handle request
3
Front controller
Controller
model
8
Return response
5
Delegate rendering of response
Create model
4
7
Return control
model
6
Render response
View template
Servlet engine (e.g. Tomcat)

Контролери


DispatcherServlet intercepts the request
Map request URI to a controller method
Execute controller method
Build response entity
Convert response entity to the type specified by request
Return response to the client/server making the request

Контролери

Анотація @Controller є конкретизацією загального стереотипу анотації @Component, яка дозволяє клас бути визнаним в якості компонента Spring.

Анотація @Controller розширює випадок @Component і позначає анотований клас, як бізнес або презентаційний шар. Коли подається запит, це повідомляє DispatcherServlet про включення класу контролера у сканування методів, відображених @RequestMapping анотацією.

Методи контролера

```
@GetMapping(path = {"/","/home-page"})
    public String homePage(){
        return "page1";
    }

    @RequestMapping(value = "/own-product", method = RequestMethod.GET)
    public ModelAndView showProduct(){
        ModelAndView result = new ModelAndView("page3");
        result.addObject("product",
                new Product("Ручка","Пуркер",300));
        return  result;
    }
```

Методи контролера

```
@RequestMapping(value = "/add-product",
        method = RequestMethod.GET)
    public String addProductForm(){
return "page4";
    }
    @RequestMapping(value = "/add-product",
        method = RequestMethod.POST,
        consumes = {"text/plain; charset=UTF-8", "application/*; charset=UTF-8"})
    public ModelAndView addProduct(Product product) {
        ModelAndView result = new ModelAndView("page3");
        result.addObject("product", new Product(product));
        return result;
    }
```

Методи контролера

```
@RequestMapping(value = "/student", method = RequestMethod.GET)
 public ModelAndView student() {

   return new ModelAndView("student", "command", new Student());
 }

 @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
   public String addStudent(Student student,
             ModelMap model) {
   model.addAttribute("name", student.getName());
   model.addAttribute("age", student.getAge());
   model.addAttribute("id", student.getId());

   return "result";
 }
```

Thymeleaf

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
    <title>My First HTML Thymeleaf Template</title>


</head>

<body>

     <h2>Hello HTML Thymeleaf Template</h2>



     <img src="../static/picture.png" th:src="@{someJavaObject.imgSrc}" />

</body>

</html>
```

Content of 'th:src' will replace content of 'src' when this template is processed by **Thymeleaf Engine**

'src' works when opening this template directly in the browser

Thymeleaf

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>Variable</title>
  </head>
  <body>
     <h1>Variables</h1>
     <h4>${variable1}</h4>
     <span th:utext="${variable1}"></span>
     <h4>${variable2}</h4>
     <span th:utext="${variable2}"></span>
  </body>
</html>
```

Thymeleaf

```
<body>
    <h1>Variable: flower, iter</h1>
    <table border="1">
        <tr>
            <th>No</th>
            <th>Flower Name</th>
        </tr>
        <tr th:each="flower, iter : ${flowers}">
            <td th:utext="${iter.count}">No</td>
            <td th:utext="${flower}">Flower Name</td>
        </tr>
    </table>
</body>
```

Thymeleaf

```
<body>
    <h1>th:with (Array)</h1>
    <th:block th:with="flowers = ${ {'Rose', 'Lily', 'Tulip'} }">
        <table border="1">
            <tr>
                <th>No</th>                <th>Flower</th>
            </tr>
            <tr th:each="flower, state : ${flowers}">
                <td th:utext="${state.count}">No</td>
                <td th:utext="${flower}">Flower</td>
            </tr>
        </table>
    </th:block>
</body>
```

Thymeleaf

```
<!-- Example -->
<div th:utext="${errorMessage} ?:  'No error!' "></div>
<!-- Example -->

<div th:object="${user}">
  ...
  <p>Age:
  <span th:text="*{age}?: '(no age specified)'">
  27</span>.
  </p>

</div>
```

Thymeleaf

```
<h1>th:if, th:unless</h1>
<div class="product-container" th:each="product : ${products}">

    <div class="img-container" th:if="${product.image}">
            <img th:src="@{|/${product.image}|}" height="70" />
     </div>
    <div class="img-container" th:unless="${product.image}">
       <img th:src="@{/no-image.png}" height="70" />
    </div>
</div>
```

Thymeleaf

```
<th:block th:include="/_menu"></th:block>
    <h2>Select Option Example 1</h2>
    <form name='f' th:object="${personForm}" method='POST'>
       <table>
          <tr>                  <td>Select Country:</td>
    <td>  <select th:field="*{countryId}">
                    <option value=""> -- </option>
                    <option th:each="country : ${countries}"
                      th:value="${country.countryId}"
                    th:utext="${country.countryName}"/></select>  </td>
          </tr>            <tr>
      <td><input name="submit" type="submit" value="submit" /></td>
          </tr>
       </table>       </form>
```

Java Persistence API (JPA)

У попередніх версіях EJB визначався рівень персистентності поєднувався з рівнем бізнес-логіки за допомогою інтерфейсу javax.ejb.EntityBean.

Під час введення EJB 3.0 рівень персистентності був відокремлений і вказаний як JPA 1.0 (Java Persistence API). Специфікації цього API були опубліковані разом зі специфікаціями JAVA EE5 11 травня 2006 р. Із використанням JSR 220.

JPA 2.0 була випущена зі специфікаціями JAVA EE6 10 грудня 2009 року як частина Java Community Process JSR 317.

JPA 2.1 був випущений із специфікацією JAVA EE7 22 квітня 2013 р. З використанням JSR 338.

JPA - це API з відкритим кодом, тому різні корпоративні постачальники, такі як Oracle, Redhat, Eclipse тощо, пропонують нові продукти, додаючи в них смак стійкості JPA.

Деякі з цих продуктів:

- **Hibernate**
- **Eclipselink**
- **Toplink**
- **Spring Data JPA**

Spring Data JPA

- Spring Data JPA, частина сімейства Spring Data, дозволяє легко впровадити сховища на основі JPA. Цей модуль стосується розширеної підтримки рівнів доступу до даних на основі JPA. Це спрощує створення Spring-програм, що використовують технології доступу до даних.
- Для виконання простих запитів, а також виконання пагінації та аудиту потрібно написати занадто багато типового коду. Spring Data JPA має на меті значно покращити реалізацію рівнів доступу до даних, зменшивши зусилля до необхідної кількості. Розробник пише власні інтерфейси сховища, включаючи користувацькі методи пошуку, а Spring забезпечує реалізацію автоматично.

Розмітка сутності

@Entity	вказує, що клас оголошується як сутність.
@Table	Ця анотація вказує на оголошення імені таблиці.
@Basic	У цій анотації явно вказуються поля без обмежень.
@Embeded	вказує властивості вбудованого класу або об'єкта
@Id	вказує первинний ключ таблиці класу.
@GeneratedValue	вказує, як атрибут ідентичності може бути ініціалізований, наприклад, автоматичний, ручний або значення взято з таблиці послідовності.
@Transient	Ця анотація вказує властивість, яка не є постійною, тобто значення ніколи не зберігається в базі даних.
@ Column	використовується, щоб вказати стовпець або атрибут для властивості.

Розмітка сутності

@SequenceGenerator	використовується для визначення значення властивості, яке вказано в анотації @GeneratedValue, створює послідовність.
@TableGenerator	використовується для визначення генератора значень для властивості, зазначеної в анотації @GeneratedValue, створює таблицю.
@AccessType	використовується для встановлення типу доступу. Якщо ви встановите @AccessType (FIELD), то буде мати місце поле для доступу. Якщо ви встановите @AccessType (PROPERTY), тоді буде відбуватися оцінка властивості.
@JoinColumn	використовується для визначення асоціації об'єктної сутності або сукупності об'єктів, використовується в асоціаціях багато до одного і один до багатьох.
@UniqueConstraint	використовується для визначення поля, унікального обмеження для первинної або вторинної таблиці.
@ColumnResult	Ця анотація посилається на назву стовпця в запиті SQL за допомогою пункту select.

Розмітка сутності

@ManyToMany	використовується для визначення зв’язку багато-до-багатьох між таблицями приєднання.
@ManyToOne	використовується для визначення зв’язку багато-до-одного між таблицями приєднання.
@OneToMany	використовується для визначення зв’язку один-до-багатьох між таблицями приєднання.
@OneToOne	використовується для визначення відношення один-до-одного між таблицями приєднання.
@NamedQueries	використовується для визначення списку іменних запитів.
@NamedQuery	використовується для визначення запиту з використанням статичного імені.

Контролери

Spring-анотація @RestController, є поєднанням @Controlleri @ResponseBody.

Ця анотація була додана під час Spring 4.0, щоб усунути надмірність оголошення @ResponseBody анотації у контролері.

У визначення інтерфейсу двох анотацій є різниця:

Інтерфейс RestController анотований @Controller і @ResponseBody замість @Component.

До всіх методів RestController застосовується анотація @ResponseBody на рівні класу.

Пакети jpa

- *Repository<T, ID extends Serializable>* interface is a marker interface that has two purposes:
  - It captures the type of the managed entity and the type of the entity's id.
  - It helps the Spring container to discover the "concrete" repository interfaces during classpath scanning.
- *CrudRepository<T, ID extends Serializable>* interface provides CRUD operations for the managed entity.
- *PagingAndSortingRepository<T, ID extends Serializable>* interface declares the methods that are used to sort and paginate entities that are retrieved from the database.
- *QueryDslPredicateExecutor<T>* interface is not a "repository interface". It declares the methods that are used to retrieve entities from the database by using QueryDsl *Predicate* objects.

Анотації для методів контролера

@RequestParam

value = “name”
required = true/false
defaultValue = “value”
produces = “type”

приклад

```
public String getNewSiteInfo(
        @RequestParam(value="site",
        required = false,
        defaultValue = "pravda.com.ua")
                String url)
        {       ...     }
```

@RequestBody

required = true/false

```
@RequestBody(required = false) Object body
```

Анотації для методів контролера

@RequestHeader

required = true/false
value = “headerName”
defaultValue = “default value”

приклад

```
public String getNewSiteInfo(
        @RequestHeader(value="site",
        required = false,
        defaultValue = "pravda.com.ua")
                String url)
        {       ...     }
```

@RequestAttribute

required = true/false
value = “attributeName”

```
@RequestAttribute(value = "attributeName", required = false) String attribute
```

@RequestPart

required = true/false
value = “attributeName”

```
@RequestPart(value = "file", required = false) MultipartFile file
```

Пакети jpa

- The *JpaRepository<T, ID extends Serializable>* interface is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.
- The *JpaSpecificationExecutor<T>* interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using *Specification<T>* objects that use the JPA criteria API.
- Швидкий старт приклад зі spring.io tutorials (https://spring.io/guides/gs/accessing-data-jpa/)

приклад

Звязок JPA-репозиторію з MVC-контроллером:

Оголошуємо інтерфейс для JPA-репозиторію

```
public interface CountryRepository extends CrudRepository<Country , Integer> {
}
```

Створюємо сервіс для роботи з ним

```
@Service("countryService")
public class CountryService {
@Autowired
CountryRepository countryRepository;
```

Приклад

```
@Transactional
public List getAllCountries() {
List countries=new ArrayList();
  Iterable countriesIterable=countryRepository.findAll();
  Iterator countriesIterator=countriesIterable.iterator();
  while(countriesIterator.hasNext())
  {   countries.add(countriesIterator.next());   }
  return countries;
      }
}
```

Приклад

Додаємо екземпляр в контроллер

```
@Controller
public class CountryController {
@Autowired
CountryService countryService;
@RequestMapping(value = "/getAllCountries", method = RequestMethod.GET,
headers = "Accept=application/json")
public String getCountries(Model model) {
    List listOfCountries = countryService.getAllCountries();
  model.addAttribute("country", new Country());
  model.addAttribute("listOfCountries", listOfCountries);
  return "countryDetails";
}
```