Spring Framework
Spring Boot

Spring Framework начинался как контейнер Dependency-Injection и превратился в полноценную экосистему.
Spring Framework предоставляет комплексную модель программирования и конфигурирования для современных корпоративных (enterprise) приложений на основе Java — на любой платформе развертывания.

Core technologies: dependency injection, events, resources, i18n, validation, data binding, type conversion.

Data Access: transactions, DAO support, JDBC, ORM, Marshalling XML.

Spring MVC and Spring WebFlux web frameworks.

IoC Container

IoC также известен как внедрение зависимостей (DI). Это процесс, в котором объекты определяют свои зависимости (то есть другие объекты, с которыми они работают) только через аргументы конструктора, аргументы фабричного метода или свойства, которые устанавливаются для экземпляра объекта после его создания или возврата из фабричного метода.
Пакеты `org.springframework.beans` и `org.springframework.context` являются основой контейнера IoC Spring Framework. Интерфейс `BeanFactory` предоставляет расширенный механизм конфигурации, способный управлять любым типом объекта. `ApplicationContext` — это подинтерфейс BeanFactory.
`BeanFactory` предоставляет конфигурации и базовые функции, а `ApplicationContext` добавляет больше функций, специфичных для enterprise приложений.
В Spring объекты, формирующие основу приложения и управляемые контейнером Spring IoC, называются **bean-компонентами**. Конфигурация Java обычно использует методы с аннотациями @Bean внутри класса с аннотацией @Configuration.
Bean компонент — это объект, который создается, собирается и управляется контейнером Spring IoC. Bean-компонент — это просто один из многих объектов приложения. Бины и зависимости между ними отражаются в метаданных конфигурации, используемых контейнером.

Интерфейс `org.springframework.context.ApplicationContext` представляет контейнер Spring IoC и отвечает за создание, настройку и сборку **bean** - компонентов. Контейнер получает инструкции о том, какие объекты создавать, настраивать и собирать, считывая метаданные конфигурации. Метаданные конфигурации могут быть представлены в формате XML, аннотациях Java или коде Java.

Например, следующий фрагмент XML объявляет два bean-компонента, InventoryService и ProductService, и внедряет InventoryService в ProductService через аргумент конструктора:

```
<bean id="inventoryService"
      class="com.example.InventoryService" />
<bean id="productService"
      class="com.example.ProductService" />
  <constructor-arg ref="inventoryService" />
</bean>
```

Однако в последних версиях Spring принято производить такие настройки в коде на Java в специальном классе-конфигураторе. Следующий Java-класс описывает эквивалентную конфигурацию:

```
@Configuration
public class ServiceConfiguration {

  @Bean
 public InventoryService inventoryService() {
    return new InventoryService();
  }

  @Bean
  public ProductService productService() {
    return new ProductService(inventoryService());
  }
}
```

Аннотация @Configuration подсказывает фреймворку Spring, что это класс конфигурации, который создает bean-компоненты для контекста Spring. Методы класса конфигурации снабжены аннотацией @Bean, указывающей, что возвращаемые ими объекты должны быть

добавлены в контекст приложения как bean-компоненты (где эти компоненты по умолчанию будут доступны по идентификаторам, совпадающим с именами определяющих их методов).

Spring Boot – это расширение для Spring Framework, предлагающее несколько улучшений. Наиболее известным из них является автоконфигурация – Spring Boot может делать обоснованные предположения о том, какие компоненты следует настроить и связать вместе, опираясь на элементы в пути поиска классов, переменные окружения и другие факторы.
Автоконфигурация, предлагаемая Spring Boot, значительно сократила объем явного описания конфигурации (с помощью XML или на Java), необходимого для создания приложения.

Bean Scopes
Spring Framework поддерживает шесть областей видимости (scopes), четыре из которых доступны, только если используется web-aware ApplicationContext.

singleton – существует один общий экземпляр компонента, и все запросы приводят к тому, что контейнер Spring возвращает этот конкретный экземпляр компонента.

```
public class Person { private String name; // standard constructor, getters and setters }

@Bean
@Scope("singleton")
 public Person personSingleton()
{ return new Person(); }

Person personSingletonA = (Person) applicationContext.getBean("personSingleton");
```

prototype - создается новый экземпляра компонента каждый раз, когда делается запрос на этот конкретный компонент.

```
@Bean
@Scope("prototype")
public Person personPrototype()
{ return new Person(); }
```

Веб-ориентированные области видимости

Request

```
public class HelloMessageGenerator
{ private String message; // standard getter and setter }

@Bean
@RequestScope
public HelloMessageGenerator requestScopedBean()
{ return new HelloMessageGenerator(); }

@Controller
 public class ScopesController
{ @Resource(name = "requestScopedBean")
HelloMessageGenerator requestScopedBean;

}
```

Session

```
@Bean @SessionScope
public HelloMessageGenerator sessionScopedBean()
{ return new HelloMessageGenerator();
 }

@Controller
 public class ScopesController
{ @Resource(name = "sessionScopedBean")
 HelloMessageGenerator sessionScopedBean;
…
}
```

application - bean ограничен уровнем ServletContext и хранится как атрибут ServletContext.

```
@Bean @ApplicationScope
public HelloMessageGenerator applicationScopedBean()
{ return new HelloMessageGenerator(); }

@Controller
 public class ScopesController
```

```
{ @Resource(name = "applicationScopedBean")
HelloMessageGenerator applicationScopedBean;
}
```

websocket – данный scope связан с жизненным циклом сеанса WebSocket.

DispatcherServlet

Spring MVC, как и многие другие веб-фреймворки, разработан на основе шаблона front controller, где центральный сервлет - DispatcherServlet, предоставляет общий алгоритм для обработки запросов, в то время как фактическая работа выполняется настраиваемыми bean-компонентами.
DispatcherServlet, как и любой другой сервлет, необходимо объявить и настроить в соответствии со спецификацией сервлета, используя конфигурацию Java или файл web.xml.
В свою очередь, DispatcherServlet использует конфигурацию Spring для обнаружения bean-компонентов, необходимых для сопоставления запросов, разрешения представлений, обработки исключений и многого другого.

Context Hierarchy

DispatcherServlet ожидает WebApplicationContext (расширение ApplicationContext) для своего конфигурирования. WebApplicationContext имеет ссылку на ServletContext и сервлет, с которым он связан.
Для многих приложений наличие одного WebApplicationContext является достаточным. Также возможно иметь контекстную иерархию (Context Hierarchy**)**, в которой один корневой WebApplicationContext используется несколькими экземплярами DispatcherServlet (или другого сервлета), каждый со своей собственной дочерней конфигурацией WebApplicationContext.
Корневой WebApplicationContext обычно содержит инфраструктурные компоненты (beans), такие как репозитории данных и бизнес-службы, которые необходимо совместно использовать для нескольких экземпляров сервлетов. Эти bean-компоненты эффективно наследуются и могут быть переопределены (то есть повторно объявлены) в специфичном для сервлета дочернем WebApplicationContext, который обычно содержит bean-компоненты, локальные для данного сервлета (рисунок 1).

DispatcherServlet делегирует управление специальным bean-компонентам для обработки запросов и вывода соответствующих ответов. Bean-компоненты, обнаруживаемые DispatcherServlet:

- HandlerMapping - сопоставляет запрос с обработчиком (handler) вместе со списком перехватчиков (interceptors) для пре- постпроцессинга. Одна из реализаций HandlerMapping — это RequestMappingHandlerMapping, который поддерживает аннотированные методы @RequestMapping);
- HandlerAdapter;
- HandlerExceptionResolver;
- ViewResolver - преобразует логические имена представлений на основе строк, возвращаемые обработчиком, в фактическое представление, с помощью которого выполняется рендеринг в ответ;
- LocaleResolver, LocaleContextResolver;
- ThemeResolver.

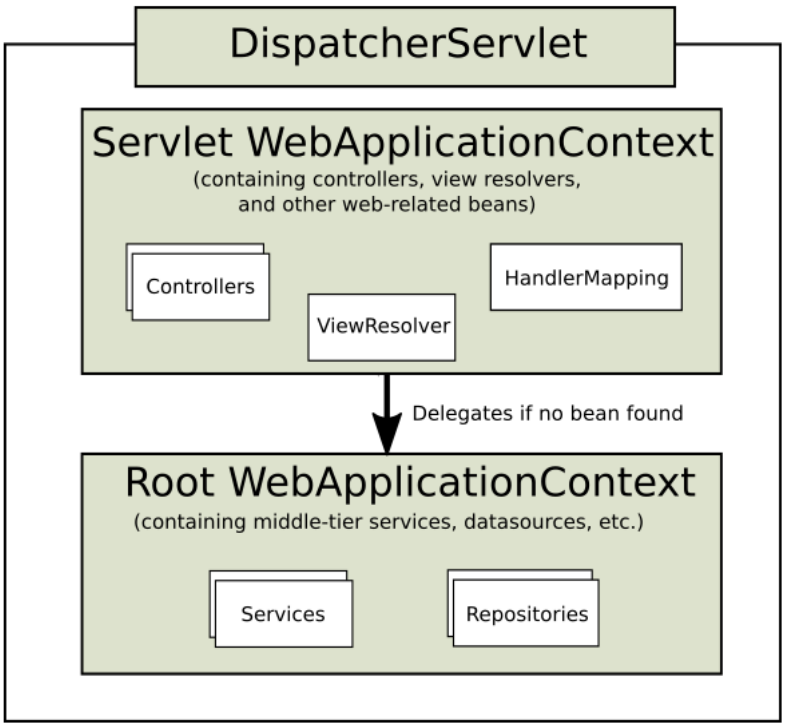

Рисунок 1

Spring Boot

Spring Boot — это проект, созданный на основе Spring Framework. Он обеспечивает более простой и быстрый способ установки, настройки и запуска как простых, так и веб-приложений.

Spring Initializr (http://start.spring.io)– это веб-приложение, помогающее создать скелетную структуру проекта Spring, которую затем можно наполнить любой функциональностью.

Spring Boot использует все модули Spring, такие как Spring MVC, Spring Data и т. д. Архитектура Spring Boot такая же, как и архитектура Spring MVC, за исключением одного: в Spring boot нет необходимости в классах DAO и DAOImpl. Поток Spring Boot:

- Создается уровень доступа к данным и выполняются операции CRUD.
- Клиент выполняет HTTP-запросы.
- Запрос поступает к контроллеру, а контроллер отображает этот запрос и обрабатывает его. После этого при необходимости вызывает логику на уровне сервисов.
- На сервисном уровне выполняется вся бизнес-логика, например, работа с данными, которые отображаются в JPA.
- Пользователю возвращается страница ответа, если не возникает ошибок

Доступ к БД

Когда Spring использует API Java Persistence (JPA) для доступа к JPA-совместимым базам данных, необходимо выбрать зависимость Spring Data JPA и зависимость для конкретного драйвера целевой базы данных, например, MySQL или PostgreSQL.

- Создание классов сущностей (@Entity)
- Репозитории. Repository — описанный в Spring Data интерфейс, играющий роль удобной абстракции для различных баз данных. `CrudRepository` можно использовать с Spring Data JPA. CrudRepository охватывает все ключевые возможности CRUD (Create, Read, Update, Delete). Также можно использовать `PagingAndSortingRepository, JpaRepository.`

`PagingAndSortingRepository` интерфейс предоставляет метод findAll(Pageable pageable), который является ключом к реализации Pagination.

При использовании Pageable создается объект Pageable с определенными свойствами:

текущий номер страницы;

размер страницы;

сортировка.

Например, необходимо отобразить первую страницу результирующего набора, отсортированную по lastName, по возрастанию, имеющую не более пяти записей на странице. Это можно сделать, используя определение PageRequest и Sort.

```
Sort sort = new Sort(new Sort.Order(Direction.ASC, "lastName"));
Pageable pageable = new PageRequest(0, 5, sort);
```

Также можно настроить репозитории для выполнения запросов, уникальных для предметной области при помощи аннотации `@Query`:

```
@Repository
public interface BikeRepository extends CrudRepository<Bike, Long>{

     @Query(value = "SELECT b FROM Bike b ORDER BY id")
     Page<Bike> findAllBikesWithPagination(Pageable pageable);

     @Query(value = "SELECT b FROM Bike b")
     List<Bike> findAllBikes(Sort sort);
}
```

Model

Модель может предоставлять атрибуты, используемые для рендеринга представлений. Чтобы предоставить представлению полезные данные, нужно добавить эти данные в объект Model. Кроме того, maps с атрибутами можно объединить с экземплярами модели:

```
@GetMapping("/showViewPage")
public String passParametersWithModel(Model model)
{ Map<String, String> map = new HashMap<>();
map.put("spring", "mvc");
model.addAttribute("message", "Baeldung");
model.mergeAttributes(map);
 return "view/viewPage"; }
```

ModelMap также используется для передачи значений для визуализации представления.

```
@GetMapping("/printViewPage")
 public String passParametersWithModelMap(ModelMap map)
{map.addAttribute("welcomeMessage", "welcome");
map.addAttribute("message", "Baeldung");
return "view/viewPage"; }
```

Интерфейс **ModelAndView** позволяет передавать сразу всю информацию, необходимую Spring MVC.

```
@GetMapping("/goToViewPage")
public ModelAndView passParametersWithModelAndView()
{ModelAndView modelAndView = new ModelAndView("view/viewPage");
 modelAndView.addObject("message", "Baeldung");
return modelAndView; }
```

Web-сервисы

- Web-сервисы — это технология создания распределенных систем, состоящих из взаимодействующих между собой программных компонент, созданных и работающих на основе различных платформ.
- Web-сервисы призваны согласовывать работу больших, состоящих из множества частей приложений, предоставляя для приложений бизнес-функции обмена данными.
- Web-сервисы могут выступать как повторно-используемые компоненты приложения, предоставляющие разнообразные сервисы — от прогноза погоды до перевода с одного языка на другой.

Типы Web-сервисов:

- SOAP Web-сервисы, ориентированные на модель RPC — вызов удаленных процедур (SOAP - Simple Object Access Protocol);
- RESTful Web-сервисы.

SOAP Web-сервисы - ориентированы на модель RPC — вызов удаленных процедур: взаимодействие с Web-сервисом производится с использованием XML-сообщений по SOAP-протоколу (Simple Object Access Protocol),
имеют интерфейсы, описанные в формате WSDL (Web Services Description Language).

Representational State Transfer (REST) - это архитектурный стиль, в котором web – сервисы рассматриваются как ресурсы и могут идентифицироваться по URI (Uniform Resource Identifiers).

Web-сервисы, разрабатываемые с помощью REST называются RESTful Web-сервисами.

RESTful Web-сервисы могут использовать различные MIME типы, но чаще всего это XML или JSON.

RESTful (Representational State Transfer) Web-сервисы :

представляют удаленные ресурсы, доступные с помощью HTTP-запросов.

RESTful Web-сервисы идентифицируются URL-адресом и обрабатывают HTTP-методы GET, PUT, POST и DELETE в ответ на запрос клиента.

Создание Spring контроллеров RESTful

Интерфейсы REST API обычно возвращают ответ в формате, ориентированном на передачу данных, таком как JSON или XML

Аннотации для различных типов HTTP-запросов перечислены в табл 1.

Таблица 1 Аннотации Spring MVC для обработки HTTP-запросов

Аннотация	Метод HTTP	Типичное применение
@GetMapping	Для обработчиков запросов HTTP GET	Чтение данных из ресурса
@PostMapping	Для обработчиков запросов HTTP POST	Создание ресурса
@PutMapping	Для обработчиков запросов HTTP PUT	Изменение содержимого ресурса
@PatchMapping	Для обработчиков запросов HTTP PATCH	Изменение содержимого ресурса
@DeleteMapping	Для обработчиков запросов HTTP DELETE	Удаление ресурса
@RequestMapping	Для универсальных обработчиков запросов	

```
@RestController
@CrossOrigin(origins = "*")
public class BikeController {
//внедрение компонента репозитория в BikeController, чтобы
//контроллер мог обращаться к нему при получении запросов через
//внешний API
     private  BikeRepository bikeRepository;
     public BikeController(BikeRepository bikeRepository) {
               this. bikeRepository = bikeRepository;
            }
```

```
@GetMapping(value="/bikes",
          produces = { MediaType.APPLICATION_JSON_VALUE })
   public List<Bike> getBikes() {
       return (List<Bike>) bikeRepository.findAll();
   }
```

CORS support in Spring Framework

Из соображений безопасности браузеры запрещают вызовы AJAX к ресурсам, находящимся за пределами текущего источника.

Совместное использование ресурсов между источниками (Cross-origin resource sharing, CORS) — это спецификация W3C, реализованная в большинстве браузеров, которая позволяет гибко указывать, какие междоменные запросы разрешены, вместо использования некоторых менее безопасных и менее мощных способов, таких как IFrame или JSONP.

Spring Framework обеспечивает встроенную поддержку CORS, предоставляя более простой способ настройки, чем обычные решения на основе фильтров.

Можно добавить к методу контроллера с аннотацией @RequestMapping аннотацию @CrossOrigin, чтобы включить CORS для него (по умолчанию @CrossOrigin разрешает все источники и методы HTTP, указанные в аннотации @RequestMapping).

Также можно включить CORS для всего контроллера.