Spring Platform

IT BOARDING

BOOTCAMP

Índice

IT BOARDING

BOOTCAMP

SPRING PLATFORM

// Repaso de conceptos

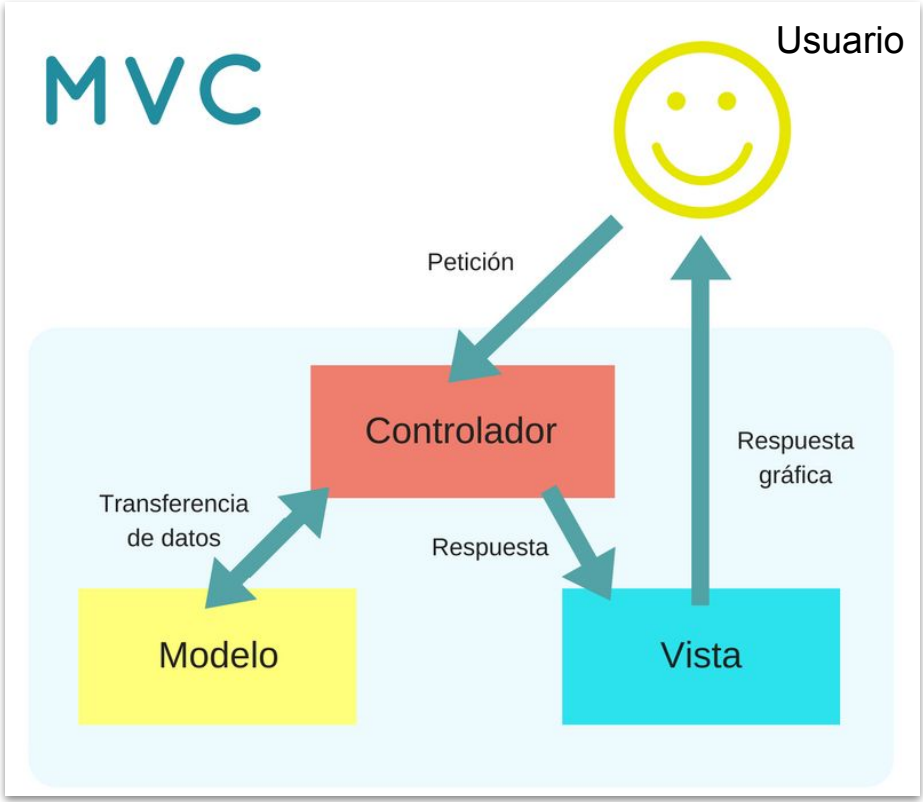
IT BOARDING

BOOTCAMP

REPASO MVC

El **Modelo-Vista-Controlador** es un patrón de arquitectura de software que separa la **lógica de negocio**, de la l**ógica de la vista** en una aplicación.

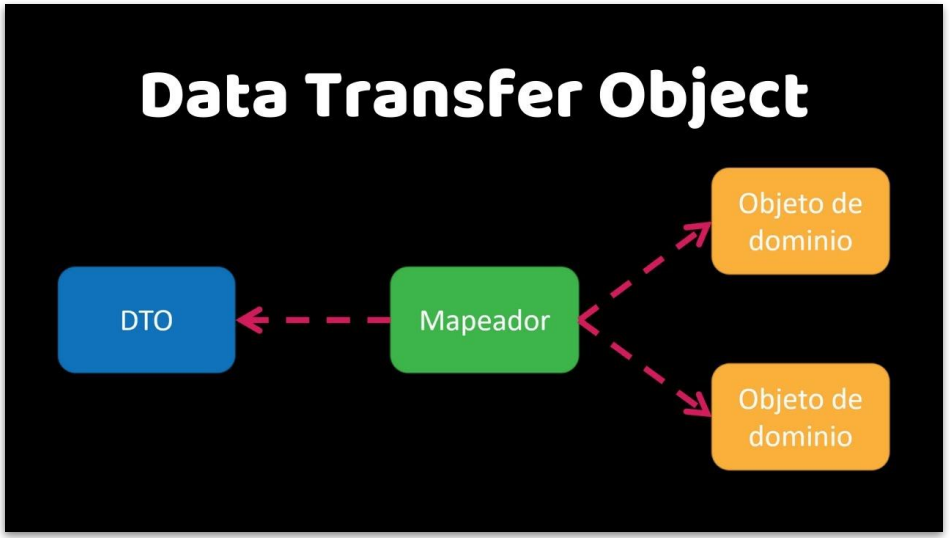
- **Modelo (@Repository, @Service):** Se encarga de los datos, generalmente (pero no obligatoriamente) consultando alguna base de datos.
- **Controlador (@RestController):** Se encarga de "controlar"; recibe las órdenes del usuario, solicita los datos al modelo y se los comunica a la vista.
- **Vista:** Es la representación visual de los datos



DTO (Data Transfer Object)

- Un DTO **es un objeto Java** utilizado para la transferencia de información.

- Mediante **@ResponseBody** le permitimos a Spring transformar un objeto java en una respuesta en formato Json.



SPRING PLATFORM

// Inyección de dependencias

IT BOARDING

BOOTCAMP

Inyección de Dependencias

- La **inyección de dependencias** es un Patrón de Diseño orientado a objetos en el que se le **suministran los objetos a una clase** en lugar de que sea ella misma quien los cree.

IoC (Inversion of Control)

- Es un p**rincipio de diseño de software** en el que **el flujo de ejecución de un programa se invierte** respecto a los métodos de programación tradicionales.

En Spring ambos conceptos trabajan de la mano, en donde se implementa un "Contenedor" que se encarga de gestionar las instancias (así como sus creaciones y destrucciones) de los objetos del usuario.

Formas de inyectar dependencias

Existen distintas formas principales mediante las cuales se puede implementar la inyección de dependencias:

- Mediante un constructor
- Mediante un método Set
- Mediante la anotación **@Autowired**

SPRING PLATFORM

// Ejemplo Práctico

IT BOARDING

BOOTCAMP

Ejemplo

Tomando como ejemplo la calculadora de metros cuadrados de clase pasada. Vamos a configurar Spring para que inyecte la clase CalculateService directamente en el RestController.

Agregar @Service a nuestra clase

```
@Service
public class CalculateService {
```

Utilizando @Autowired vs Constructor

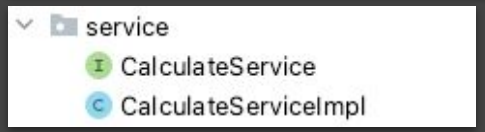
```
@RestController
public class CalculateRestController {
  @Autowired
  private CalculateService calculateService;

  @PostMapping("/calculate")
  public HouseResponseDTO calculate(@RequestBody HouseDTO house){
    return calculateService.calculate(house);
  }
}
```

```
@RestController
public class CalculateRestController {
  private final CalculateService calculateService;

  public CalculateRestController(CalculateService calculateService) {
    this.calculateService = calculateService;
  }

  @PostMapping("/calculate")
  public HouseResponseDTO calculate(@RequestBody HouseDTO house){
    return calculateService.calculate(house);
  }
}
```

Interfaces e implementaciones

Una mejor práctica para poder separar las capas seria que la comunicación entre ellas sea mediante Interfaces y no clases concretas.

```
@Service
public class CalculateServiceImpl implements CalculateService {
```




```
public interface CalculateService {
  HouseResponseDTO calculate(HouseDTO house);
}
```

"Esta práctica permite poder cambiar las implementaciones sin la necesidad de modificar el RestController o quien utilice esta dependencias" (inversión de dependencia o control)

Creando un repositorio

Ahora vamos a obtener el precio por metro cuadrado dependiendo de donde se encuentra la casa, obteniendo la información de un JSON

```
[
 {
   "location": "Palermo",
   "price": 1000
 },
 {
   "location": "Belgrano",
   "price": 1100
 },
 {
   "location": "Recoleta",
   "price": 900
 },
 {
   "location": "Puerto Madero",
   "price": 2000
 }
]
```

Creando repositorio

Crear package repositories y una interfaz dentro llamada PriceRepository con el siguiente método:

```
public interface PriceRepository {
  PriceDTO findPriceByLocation(String location);
}
```

```
public class PriceDTO {
  private String location;
  private Integer price;

  public PriceDTO() {
  }

  public String getLocation() {
    return location;
  }

  public void setLocation(String location) {
    this.location = location;
  }

  public Integer getPrice() {
    return price;
  }

  public void setPrice(Integer price) {
    this.price = price;
  }
}
```

Creando repositorio

La implementación va a leer el archivo JSON utilizando ObjectMapper y va a retornar el PriceDTO dependiendo el nombre.

Al HouseDTO, tenemos que agregarle el atributo **String location;**

```java
@Repository
public class PriceRepositoryImpl implements PriceRepository{
  @Override
  public PriceDTO findPriceByLocation(String location) {
    List<PriceDTO> priceDTOS = null;
    priceDTOS = loadDataBase();
    PriceDTO result = null;
    if (priceDTOS != null){
      Optional<PriceDTO> item = priceDTOS.stream()
          .filter(priceDTO -> priceDTO.getLocation().equals(location))
          .findFirst();
      if (item.isPresent())
        result = item.get();
    }
    return result;
  }

  private List<PriceDTO> loadDataBase() {
    File file = null;
    try {
      file = ResourceUtils.getFile( resourceLocation: "classpath:prices.json");
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
    ObjectMapper objectMapper = new ObjectMapper();
    TypeReference<List<PriceDTO>> typeRef = new TypeReference<>() {};
    List<PriceDTO> priceDTOS = null;
    try {
      priceDTOS = objectMapper.readValue(file, typeRef);
    } catch (IOException e) {
      e.printStackTrace();
    }
    return priceDTOS;
  }
}
```

Usando repositorio

```java
@Service
public class CalculateServiceImpl implements CalculateService {
  private final PriceRepository priceRepository;

  public CalculateServiceImpl(PriceRepository priceRepository) {
    this.priceRepository = priceRepository;
  }

  private int calculatePrice(Integer result, String location) {
    PriceDTO priceByLocation = priceRepository.findPriceByLocation(location);
    Integer price = priceByLocation != null ? priceByLocation.getPrice() : 800;
    return result * price;
  }

  public HouseResponseDTO calculate(HouseDTO house) {
    HouseResponseDTO response = new HouseResponseDTO(house);
    calculateRoomSquareFeet(house, response);
    response.setPrice(calculatePrice(response.getSquareFeet(), house.getLocation()));
    return response;
  }
```

// ¿Dudas?
¿Preguntas?

Diccionario/Resumen de Anotaciones de esta clase

- **@Service:** Identifica cap de servicios.
- **@Repository:** Anotación para identificar el controlador de un servicio REST
- **@Autowired:** Inyecta instancias de objetos dependientes automáticamente.
- **@Getter / @Setter**: automatiza la generación de esos métodos.