

# Spring для начинающих



**[zaurtregulov@gmail.com](mailto:zaurtregulov@gmail.com)**

# Для кого предназначен данный курс?

- Для людей, совсем ничего не знающих о Spring и Hibernate
- Для людей, которые хотят расширить знания о Spring и Hibernate и закрепить свои знания различными примерами

# Чем необходимо обладать для начала просмотра курса?

- Базовыми знаниями языка программирования Java
- **ЖЕЛАНИЕМ** изучить Spring, Hibernate и Spring Boot

Простые  
объяснения

+

Простые  
примеры

+

Много  
практики

=

Успешное  
обучение

# Содержание курса

IoC и DI

Spring MVC + Hibernate + AOP

AOP

Spring REST

Hibernate

Spring Security

Spring MVC

Spring Boot

# Spring для начинающих

- Уроки стали короткими
- Уроки сохранили эффективность
- Стало ещё больше примеров

Спрашивайте и Помогайте

Не забудьте оценить



# Spring

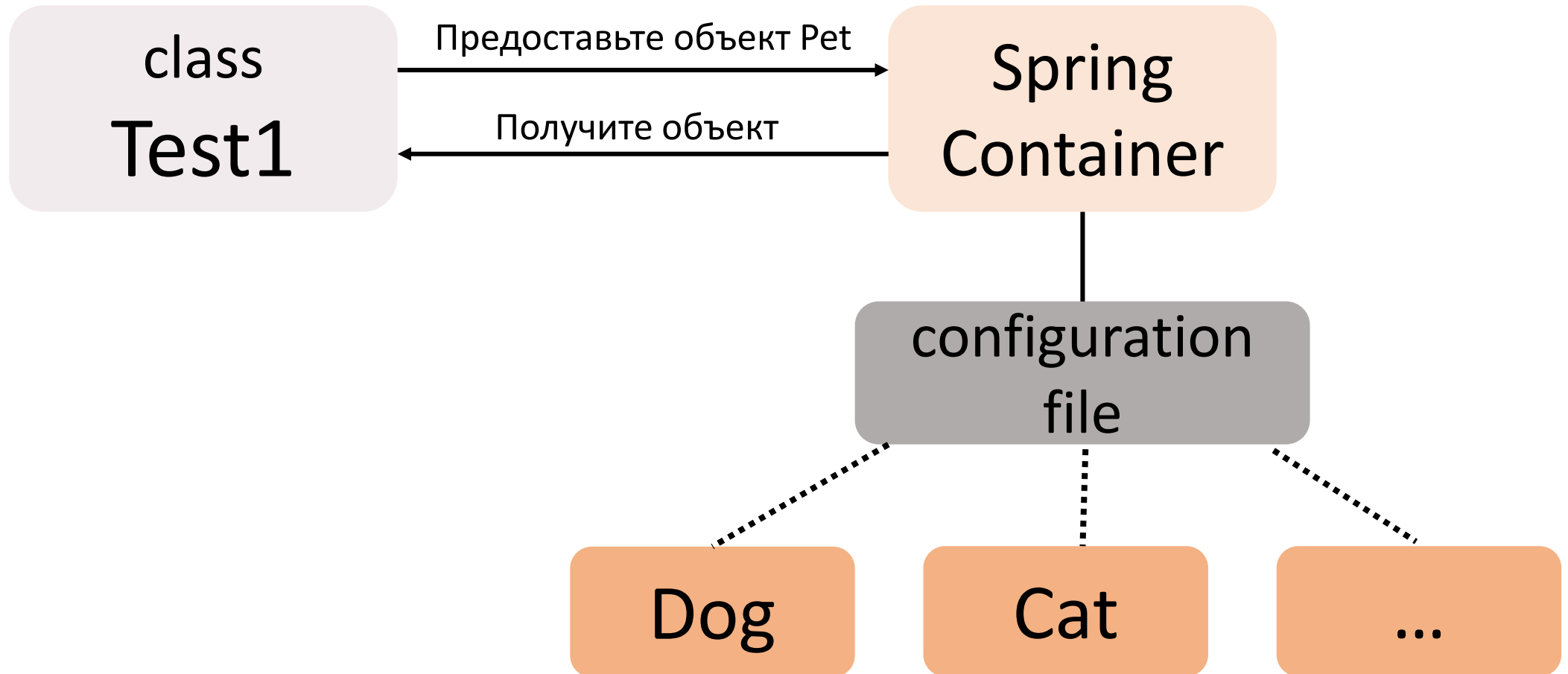
Spring – это фреймворк, предназначенный для более быстрого и простого построения Java приложений

Год создания: 2004

Последняя версия (на 27.10.2020) – **Spring 5**

Официальный сайт: [www.spring.io](https://www.spring.io)

# Inversion of Control



# Inversion of Control

Основные функции, которые выполняет  
Spring Container:

- IoC – инверсия управления  
Создание и управление объектами
- DI – Dependency Injection  
Внедрение зависимостей

IoC – аутсорсинг создания и управления объектами. Т.е. передача программистом прав на создание и управление объектами Spring-у.



# Inversion of Control

Способы конфигурации Spring Container:

- XML file (устаревший способ)
- Annotations + XML file (современный способ)
- Java code (современный способ)

# Inversion of Control

Конфигурация XML файла:

```
<bean id = "myPet"  
      class = "ioc.Cat">  
</bean>
```

- id – идентификатор бина
- class – полное имя класса

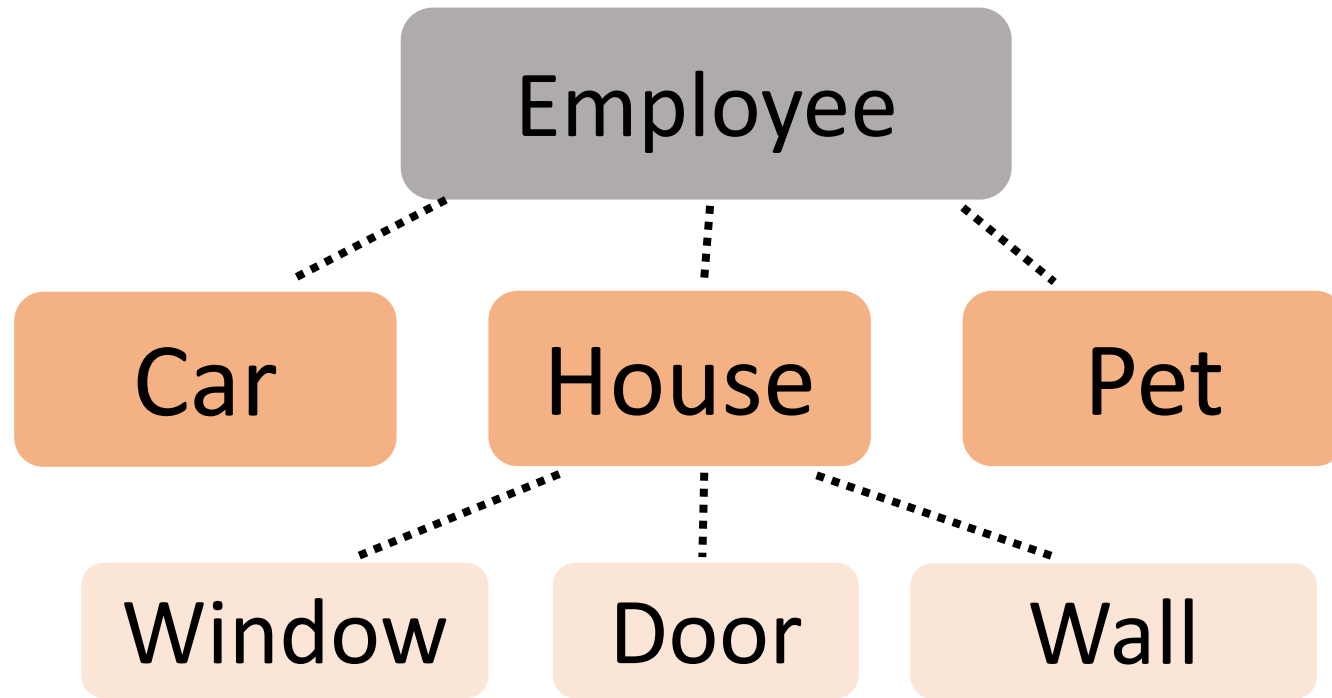
# Inversion of Control

Spring Bean (или просто bean) – это объект, который создаётся и управляется Spring Container

ApplicationContext представляет собой Spring Container. Поэтому для получения бина из Spring Container нам нужно создать ApplicationContext.

```
ClassPathXmlApplicationContext context =  
    new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml" );  
  
Pet pet = context.getBean( name: "myPet", Pet.class );
```

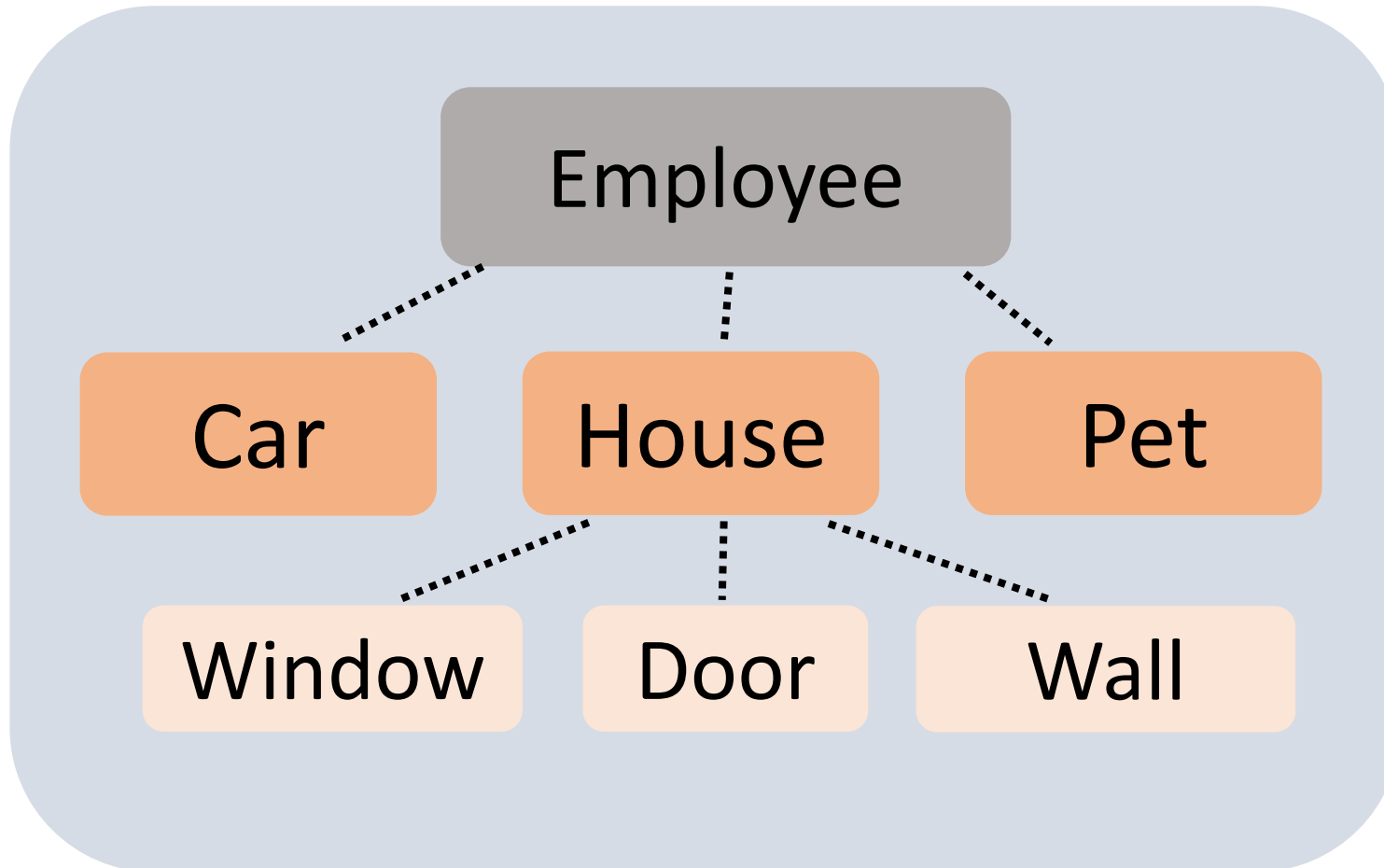
# Dependency Injection



```
Employee emp = new Employee();  
Car car = new Car();           emp.setCar(car);  
House house = new House();  
Door door = new Door();        house.setDoor(door);
```

# Dependency Injection

Spring Container



# Dependency Injection

DI – аутсорсинг добавления/внедрения зависимостей. DI делает объекты нашего приложения слабо зависимыми друг от друга.

Способы внедрения зависимостей:

- С помощью конструктора
- С помощью сеттеров
- Autowiring

# Dependency Injection

DI с помощью конструктора:

```
<bean id = "myPet"  
      class = "ioc.Cat">  
</bean>
```

```
<bean id = "myPerson"  
      class="ioc.Person">  
  <constructor-arg ref="myPet"/>  
</bean>
```

За кулисами:

```
Cat myPet = new Cat();
```

```
Person myPerson = new Person(myPet);
```

- constructor-arg – аргумент конструктора
- ref – ссылка на bean id

# Dependency Injection

DI с помощью сеттера:

```
<bean id = "myPet"  
      class = "ioc.Cat">  
</bean>
```

```
<bean id = "myPerson"  
      class="ioc.Person">  
  <property name="pet" ref="myPet"/>  
</bean>
```

За кулисами:

```
Cat myPet = new Cat();
```

```
Person myPerson = new Person();  
myPerson.setPet(myPet);
```

Первая буква в слове «pet» становится заглавной и в начало слова добавляется «set». После чего вызывается получившийся метод.



# Dependency Injection

Внедрение строк и других значений:

```
<bean id = "myPerson"  
      class="ioc.Person">  
  <property name="surname" value="Tregulov"/>  
  <property name="age" value="33"/>  
</bean>
```

За кулисами:

```
Person myPerson = new Person();  
myPerson.setSurname("Tregulov");  
myPerson.setAge(33);
```

- value – значение, которое мы хотим присвоить.

# Dependency Injection

Внедрение строк и других значений из properties файла:

```
person.surname = Tregulov  
person.age = 33
```

```
<context:property-placeholder location="classpath:myApp.properties"/>
```

```
<bean id = "myPerson"  
      class="ioc.Person">  
  <property name="surname" value="${person.surname}"/>  
  <property name="age" value="${person.age}"/>  
</bean>
```

За кулисами:

```
Person myPerson = new Person();  
myPerson.setSurname("Tregulov");  
myPerson.setAge(33);
```

# IoC & DI

IoC – аутсорсинг создания и управления объектами. Т.е. передача программистом прав на создание и управление объектами Spring-у.

DI – аутсорсинг добавления/внедрения зависимостей. DI делает объекты нашего приложения слабо зависимыми друг от друга.

Большое количество программистов используют эти термины как взаимозаменяемые.

# Bean scope

Score (область видимости) определяет:

- жизненный цикл бина
- возможное количество создаваемых бинов

Разновидности bean scope:

**singleton**

**prototype**

request

session

global-session

# Bean scope

singleton – дефолтный scope.

- такой бин создаётся сразу после прочтения Spring Container-ом конфиг файла.
- является общим для всех, кто запросит его у Spring Container-а.
- подходит для stateless объектов.

```
<bean id="myPet"  
      class="ioc.Dog"  
      scope="singleton">  
</bean>
```

```
Dog myDog = context.getBean(name: "myPet", Dog.class);
```

```
Dog yourDog = context.getBean(name: "myPet", Dog.class);
```

Spring Container

Dog

# Bean scope

## prototype

- такой бин создаётся только после обращения к Spring Container-у с помощью метода `getBean`.
- для каждого такого обращения создаётся новый бин в Spring Container-е.
- подходит для stateful объектов.

```
<bean id="myPet"  
      class="ioc.Dog"  
      scope="prototype">
```

```
</bean>
```

```
Dog myDog = context.getBean(name: "myPet", Dog.class);
```

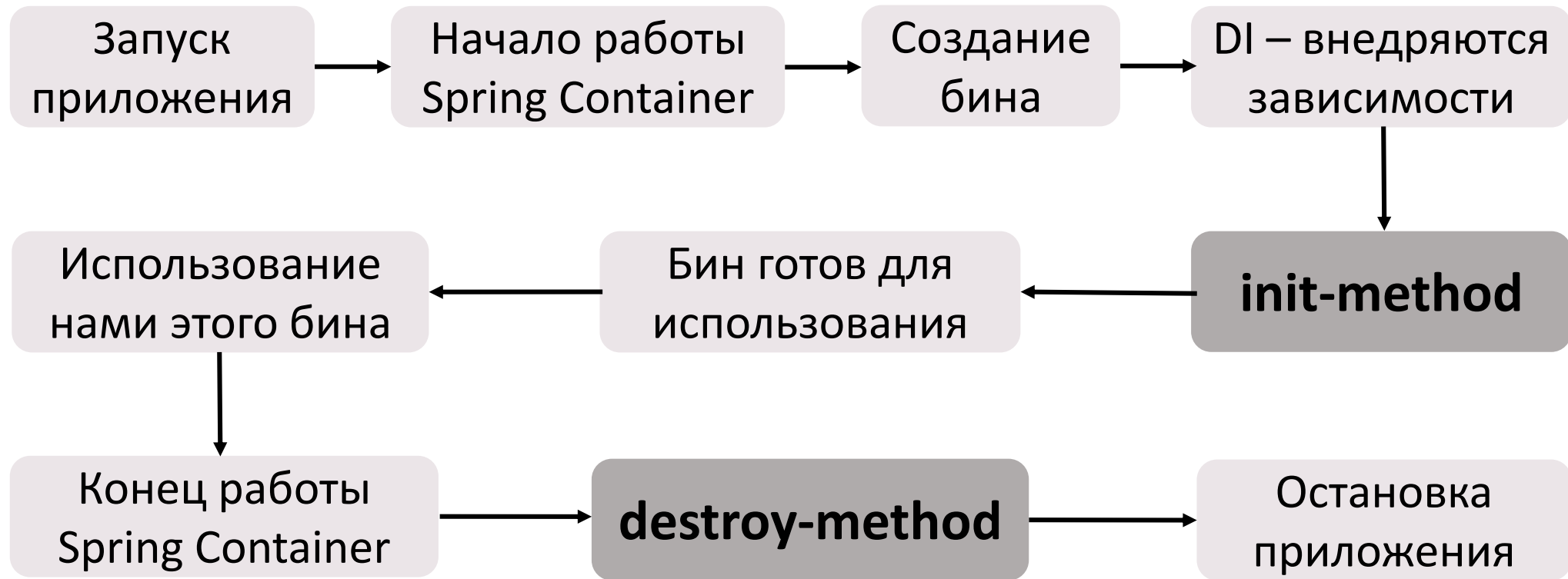
```
Dog yourDog = context.getBean(name: "myPet", Dog.class);
```

Spring Container

Dog

Dog

# Жизненный цикл бина



Чаще всего **init-method** используется для открытия или настройки каких-либо ресурсов, например баз данных, стримов и т.д.  
**destroy-method** чаще всего используется для их закрытия.

# Методы init и destroy

```
public void init() {  
    System.out.println("Class Dog: init method");  
}  
  
public void destroy() {  
    System.out.println("Class Dog: destroy method");  
}
```

```
<bean id="myPet"  
    class="ioc.Dog"  
    init-method="init"  
    destroy-method="destroy">  
</bean>
```



# Методы `init` и `destroy`

У данных методов `access modifier` может быть любым

У данных методов `return type` может быть любым. Но из-за того, что возвращаемое значение мы никак не можем использовать, чаще всего `return type` – это `void`.

Называться данные методы могут как угодно.

В данных методах не должно быть параметров.

# Методы `init` и `destroy`

Если у бина `scope = prototype`, то:

- `init-method` будет вызываться для каждого новосозданного бина.
- для этого бина `destroy-method` вызываться не будет
- программисту необходимо самостоятельно писать код для закрытия/освобождения ресурсов, которые были использованы в бине

# Конфигурация с помощью аннотаций

Аннотации – это специальные комментарии/метки/метаданные, которые нужны для передачи определённой информации.

Конфигурация с помощью аннотаций более короткий и быстрый способ, чем конфигурация с помощью XML файла.

Процесс состоит из 2-х этапов:

1. сканирование классов и поиск аннотации `@Component`
2. Создание (регистрация) бина в Spring Container-e

# Конфигурация с помощью аннотаций

```
<context:component-scan base-package="spring_introduction"/>
```

```
import org.springframework.stereotype.Component;
```

```
@Component("catBean")
```

```
public class Cat implements Pet {
```

```
    public Cat() {
```

```
        System.out.println("Cat bean is created");
```

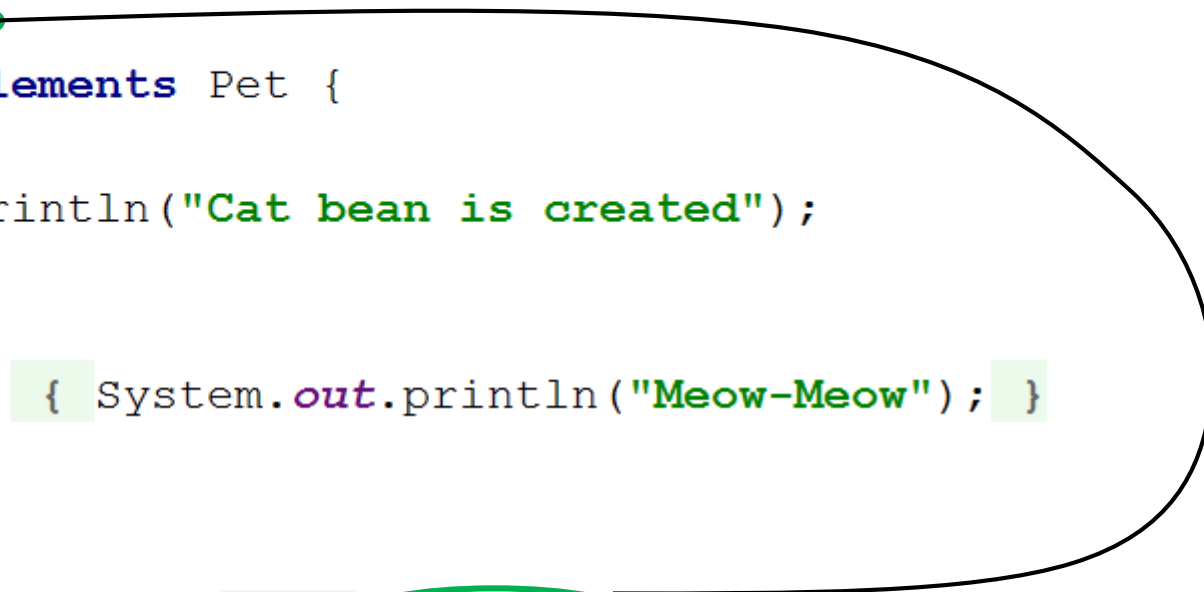
```
    }
```

```
    @Override
```

```
    public void say() { System.out.println("Meow-Meow"); }
```

```
}
```

```
Cat myCat = context.getBean(name: "catBean", Cat.class);
```



# Конфигурация с помощью аннотаций

Если к аннотации `@Component` не прописать `bean id`, то бину будет назначен дефолтный `id`.

Дефолтный `bean id` получается из имени класса, заменяя его первую заглавную букву на прописную.

```
@Component
class Cat {
    }
    cat

@Component
class FavoriteSong{
    }
    favoriteSong

@Component
class SQLTest{
    }
    SQLTest
```

# @Autowired

Для внедрения зависимостей с помощью аннотаций используется аннотация @Autowired

Типы autowiring-а или где мы можем использовать данный DI:

- Конструктор
- Сеттер
- Поле

# @Autowired

Процесс внедрения зависимостей при использовании @Autowired такой:

1. Сканирование пакета, поиск классов с аннотацией @Component
2. При наличии аннотации @Autowired начинается поиск подходящего по типу бина

Далее ситуация развивается по одному из сценариев:

- Если находится 1 подходящий бин, происходит внедрение зависимости;
- Если подходящих по типу бинов нет, то выбрасывается исключение;
- Если подходящих по типу бинов больше одного, тоже выбрасывается исключение.

# @Autowired

```
@Component("catBean")  
public class Cat implements Pet {
```

## Constructor injection

```
@Component("personBean")  
public class Person {  
    private Pet pet;  
  
    @Autowired  
    public Person(Pet pet) {  
        this.pet = pet;  
    }  
}
```

## Setter injection

```
@Autowired  
public void setPet(Pet pet) {  
    this.pet = pet;  
}
```

## Field injection

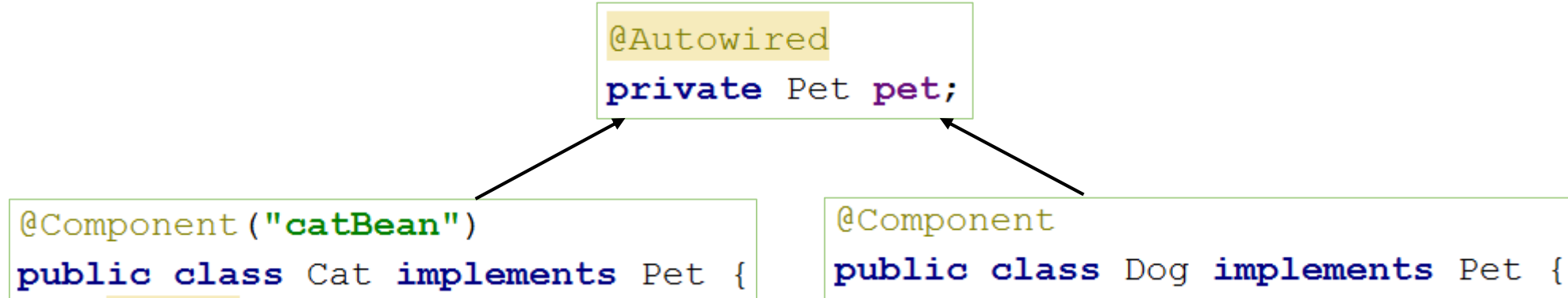
```
@Autowired  
private Pet pet;
```

## Any method injection

```
@Autowired  
public void anyMethodName(Pet pet) {  
    this.pet = pet;  
}
```

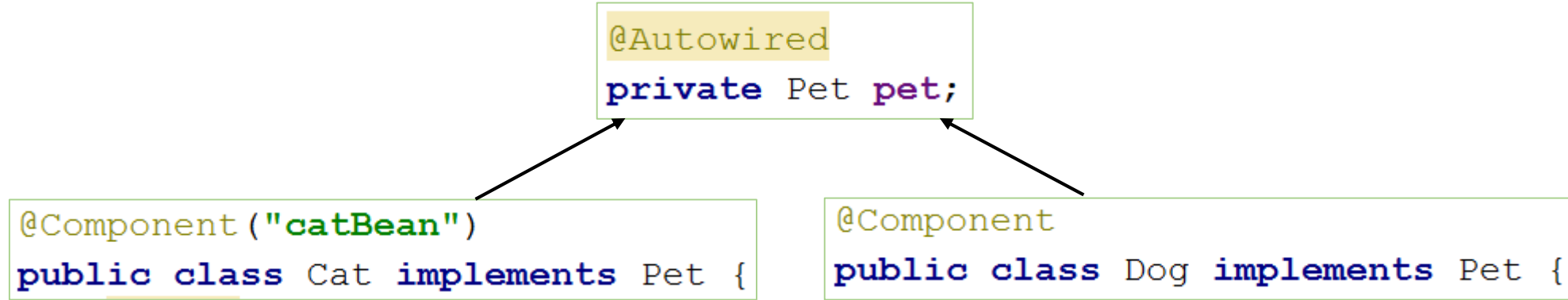


# @Qualifier



Если при использовании `@Autowired` подходящих по типу бинов больше одного, то выбрасывается исключение. Предотвратить выброс данного исключения можно конкретно указав, какой бин должен быть внедрён. Для этого и используют аннотацию `@Qualifier`.

# @Qualifier



Field

Setter

Constructor

```
@Autowired
@Qualifier("dog")
private Pet pet;
```

```
@Autowired
@Qualifier("dog")
public void setPet(Pet pet) {
    this.pet = pet;
}
```

```
@Autowired
public Person(@Qualifier("dog") Pet pet) {
    this.pet = pet;
}
```

# @Value

Для внедрения строк и других значений можно использовать аннотацию @Value.

В этом случае в сеттерах нет необходимости, как это было при конфигурации с помощью XML файла.

Hardcoded вариант

```
@Value("Tregulov")  
private String surname;  
@Value("33")  
private int age;
```

Вариант с properties файлом

```
<context:component-scan base-package="spring_introduction"/>  
<context:property-placeholder location="classpath:myApp.properties"/>
```

```
@Value("${person.surname}")  
private String surname;  
@Value("${person.age}")  
private int age;
```

# Bean scope

singleton – дефолтный scope.

- такой бин создаётся сразу после прочтения Spring Container-ом конфиг файла.
- является общим для всех, кто запросит его у Spring Container-a.
- подходит для stateless объектов.

prototype

- такой бин создаётся только после обращения к Spring Container-у с помощью метода `getBean`.
- для каждого такого обращения создаётся новый бин в Spring Container-e.
- подходит для stateful объектов.

```
@Component
@Scope("singleton")
public class Dog implements Pet {
```

```
@Component
@Scope("prototype")
public class Dog implements Pet {
```

# Методы `init` и `destroy`

У данных методов `access modifier` может быть любым

У данных методов `return type` может быть любым. Но из-за того, что возвращаемое значение мы никак не можем использовать, чаще всего `return type` – это `void`.

Называться данные методы могут как угодно.

В данных методах не должно быть параметров.

# Методы init и destroy

Если у бина scope = prototype, то:

- init-method будет вызываться для каждого новосозданного бина.
- для этого бина destroy-method вызываться не будет
- программисту необходимо самостоятельно писать код для закрытия/освобождения ресурсов, которые были использованы в бине

# @PostConstruct и @PreDestroy

```
@PostConstruct
public void init() {
    System.out.println("Class Dog: init method");
}

@PreDestroy
public void destroy() {
    System.out.println("Class Dog: destroy method");
}
```

# Конфигурация Spring Container-а с помощью Java кода. Способ 1

```
@Configuration
@ComponentScan("spring_introduction")
public class MyConfig {
}
```

Аннотация `@Configuration` означает, что данный класс является конфигурацией.

С помощью аннотации `@ComponentScan` мы показываем, какой пакет нужно сканировать на наличие бинов и разных аннотаций.

```
AnnotationConfigApplicationContext context =
    new AnnotationConfigApplicationContext(MyConfig.class);
```

При использовании конфигурации с помощью Java кода, Spring Container будет представлен классом `AnnotationConfigApplicationContext`



# Конфигурация Spring Container-а с помощью Java кода. Способ 2

```
@Configuration
public class MyConfig {
    @Bean
    public Pet cat() {
        return new Cat();
    }

    @Bean
    public Person myPerson() {
        return new Person(cat());
    }
}
```

- Данный способ не использует сканирование пакета и поиск бинов. Здесь бины описываются в конфиг классе.
- Данный способ не использует аннотацию @Autowired. Здесь зависимости прописываются вручную.
- Название метода – это bean id.
- Аннотация @Bean перехватывает все обращения к бину и регулирует его создание.

# Аннотация @PropertySource

```
@Value("${person.surname}")  
private String surname;  
@Value("${person.age}")  
private int age;
```

```
@Configuration  
@PropertySource("classpath:myApp.properties")  
public class MyConfig {
```

Аннотация @PropertySource указывает на property файл откуда мы можем использовать значения для полей

# Aspect Oriented Programming

Method `addBook(String bookName, int personId)`

Логирование

Проверка прав доступа

Основная логика метода

# Aspect Oriented Programming



# Aspect Oriented Programming

Проблемы, с которыми мы сталкиваемся:

- Переплетение бизнес-логики со служебным функционалом (Code tangling). Метод становится громоздким, и его основной функционал сразу не заметно.
- Разбросанность служебного функционала по всему проекту (Code scattering). При необходимости что-то изменить в служебном функционале, мы должны будем делать изменения во всех классах.

# Aspect Oriented Programming

AOP – парадигма программирования, основанная на идее разделения основного и служебного функционала. Служебный функционал записывается в Aspect-классы.

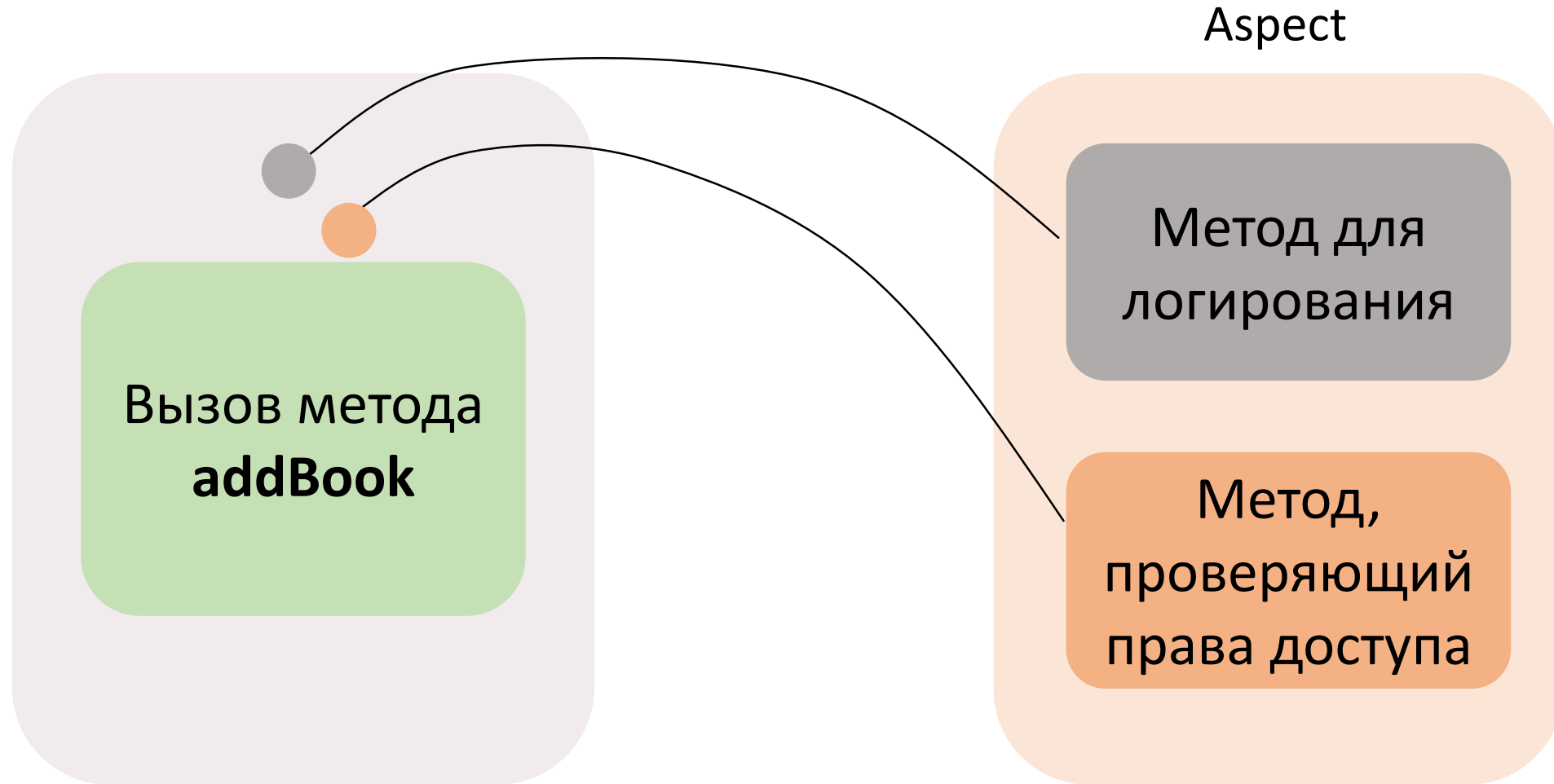
В основе Aspect заключена сквозная логика (cross-cutting logic).

# Aspect Oriented Programming

К сквозному функционалу относят:

- Логирование
- Проверка прав (security check)
- Обработка транзакций
- Обработка исключений
- Кэширование
- И т.д.

# Aspect Oriented Programming





# Aspect Oriented Programming

Main класс

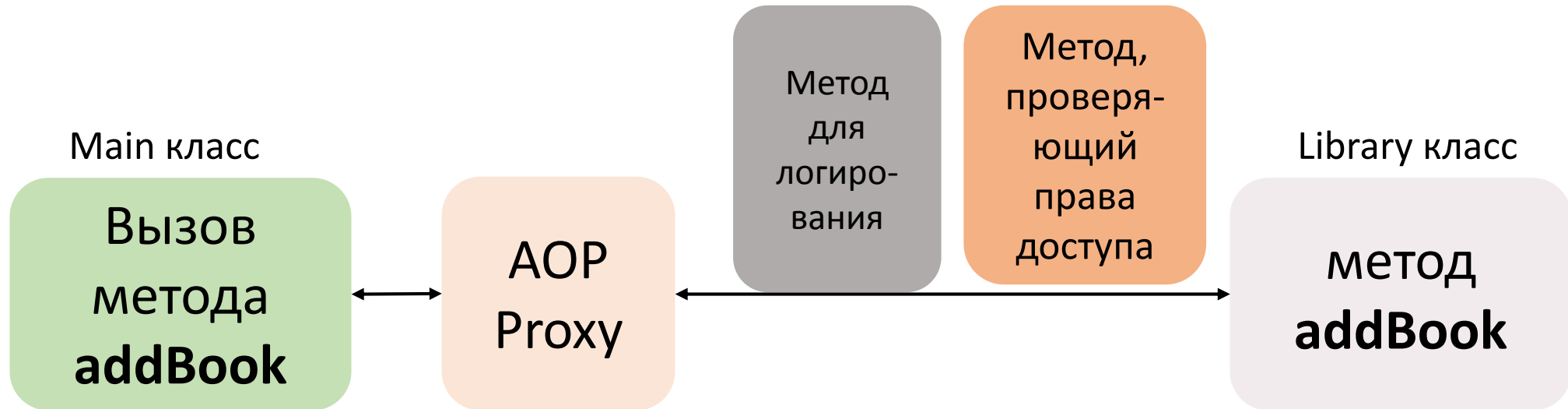
Вызов  
метода  
**addBook**

Library класс

метод  
**addBook**



# Aspect Oriented Programming



# Aspect Oriented Programming

## Плюсы AOP:

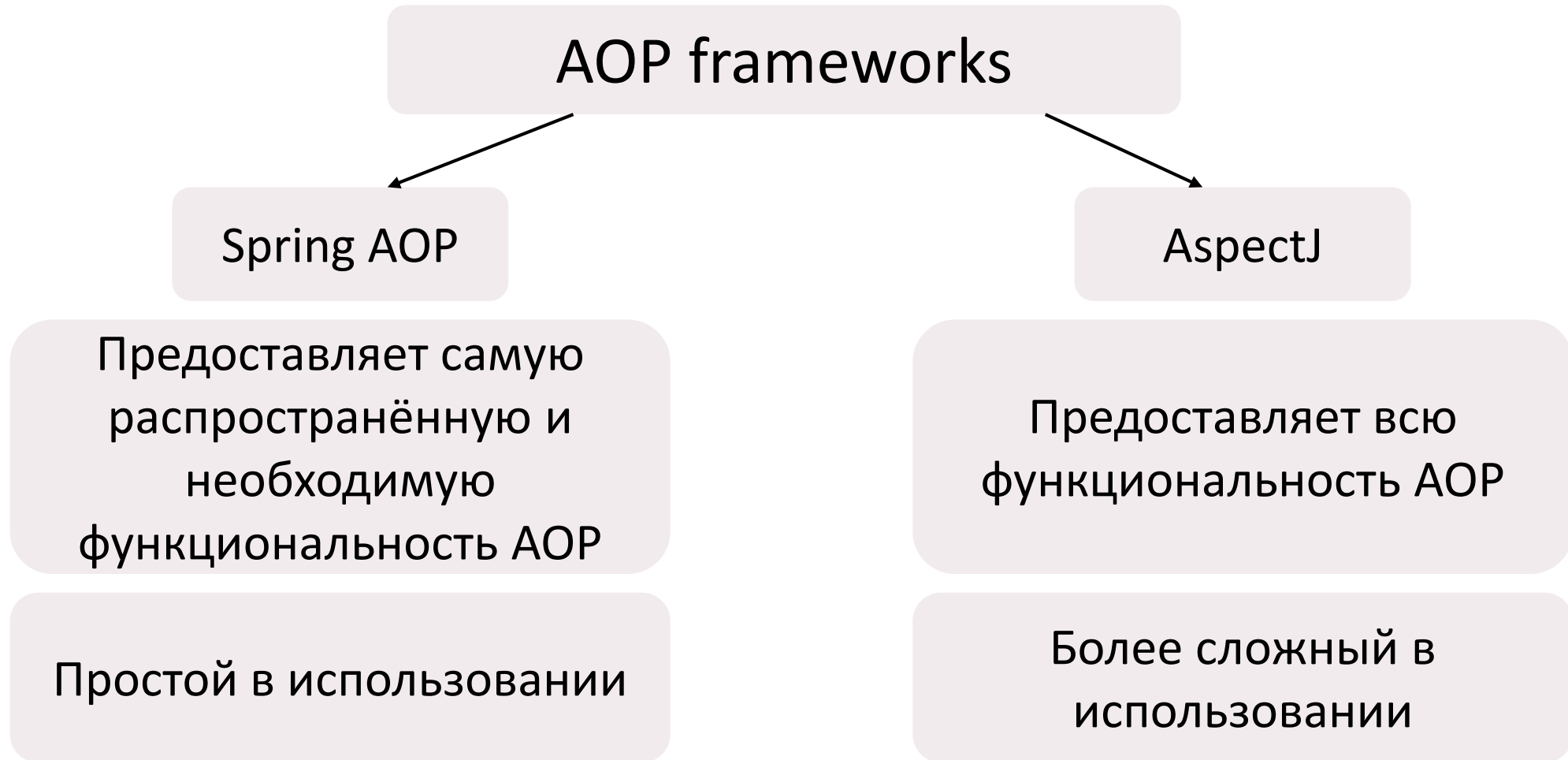
- Сквозной функционал сосредоточен в 1-м или нескольких обособленных классах. Это позволяет легче его изменять.
- Становится легче добавлять новые сквозные работы для нашего основного кода или имеющиеся сквозные работы для новых классов. Это достигается благодаря конфигурации аспектов.
- Бизнес-код приложения избавляется от сквозного кода, становится меньше и чище. Работать с ним становится легче.

# Aspect Oriented Programming

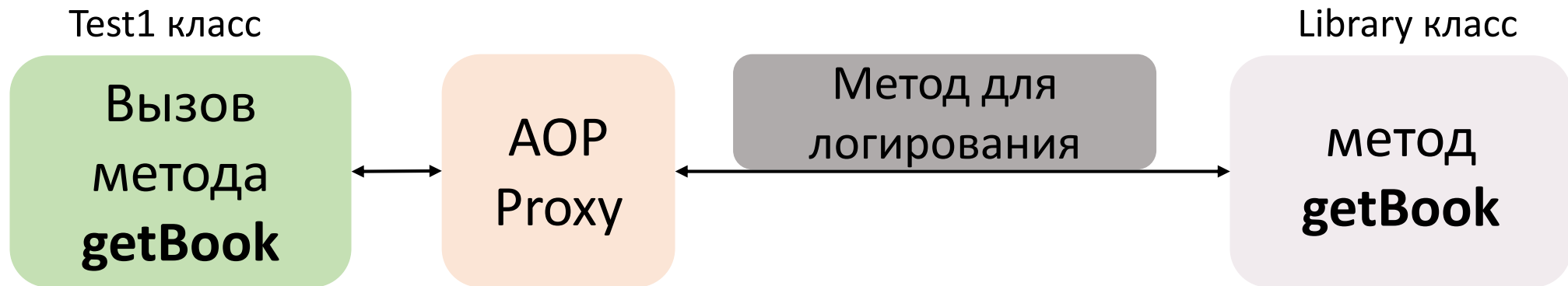
Минус AOP:

- Дополнительное время на работу аспектов

# Aspect Oriented Programming



# Aspect Oriented Programming



# Aspect

```
@Configuration  
@ComponentScan("aop")  
@EnableAspectJAutoProxy  
public class MyConfig {  
}
```

@EnableAspectJAutoProxy позволяет нам за кулисами использовать Spring AOP Proxy

```
@Component  
@Aspect  
public class LoggingAspect {  
}
```

@Aspect говорит о том, что это не простой класс, а Aspect. Поэтому к данному классу Spring будет относиться по другому.

Aspect – это класс, отвечающий за сквозную функциональность.

# Advice Типы

- Before – выполняется до метода с основной логикой
- After returning – выполняется только после нормального окончания метода с основной логикой
- After throwing – выполняется после окончания метода с основной логикой только, если было выброшено исключение
- After/ After finally – выполняется после окончания метода с основной логикой
- Around – выполняется до и после метода с основной логикой



# Advice и Pointcut

```
public void getBook() {  
    System.out.println("Мы берём книгу ");  
}
```

```
@Component  
@Aspect  
public class LoggingAspect {  
  
    @Before("execution(public void getBook())")  
    public void beforeGetBookAdvice() {  
        System.out.println("beforeGetBookAdvice: попытка " +  
            "получить книгу");  
    }  
}
```

Advice – метод, который находится в Aspect-е и содержит сквозную логику. Advice определяет, что и когда должно происходить.

В идеале Advice должен быть небольшим и быстро работающим.

Pointcut – выражение, описывающее где должен быть применён Advice.

# Pointcut

Pointcut – выражение, описывающее где должен быть применён Advice.

Spring AOP использует AspectJ Pointcut expression language. Т.е. определённые правила в написании выражений для создания Pointcut

```
execution( modifiers-pattern? return-type-pattern declaring-type-pattern?  
method-name-pattern(parameters-pattern) throws-pattern? )
```

# Pointcut

```
execution(public void getBook())
```

Соответствует методу без параметров, где бы он ни находился с access modifier = public, return type = void и именем = getBook

```
execution(public void aop.UniversityLibrary.getBook())
```

Соответствует методу без параметров, из класса UniversityLibrary с access modifier = public, return type = void и именем = getBook

```
execution(public void get*())
```

Соответствует методу без параметров, где бы он ни находился с access modifier = public, return type = void и именем, начинающимся на «get»

```
execution(* getBook())
```

Соответствует методу без параметров, где бы он ни находился с любым access modifier, любым return type и именем = getBook

```
execution(* *())
```

Соответствует методу без параметров, где бы он ни находился с любым access modifier, любым return type и любым именем

# Pointcut

```
execution(public void getBook(String))
```

Соответствует методу с параметром String, где бы он ни находился с access modifier = public, return type = void и именем = getBook

```
execution(public void getBook(*))
```

Соответствует методу с любым одним параметром, где бы он ни находился с access modifier = public, return type = void и именем = getBook

```
execution(public void getBook(..))
```

Соответствует методу с любым количеством любого типа параметров, где бы он ни находился с access modifier = public, return type = void и именем = getBook

# Pointcut

```
execution(public void getBook(aop.Book, ..))
```

Соответствует методу, первым параметром которого является aop.Book, а дальше может идти 0 и больше параметров любого типа, где бы этот метод ни находился с access modifier = public, return type = void и именем = getBook

```
execution(* * (..))
```

Соответствует методу с любым количеством любого типа параметров, где бы он ни находился с любым access modifier, любым return type и любым именем

# Объявление Pointcut

Для того, чтобы не пользоваться copy-paste когда для нескольких Advice-ов подходит один и тот же Pointcut, есть возможность объявить данный Pointcut и затем использовать его несколько раз.

```
@Pointcut("pointcut_expression")  
private void pointcut_reference() {}
```

```
@Before("pointcut_reference() ")  
public void advice_name() {    some code    }
```

# Объявление Pointcut

## Плюсы объявления Pointcut:

- Возможность использования созданного Pointcut для множества Advice-ов
- Возможность быстрого изменения Pointcut expression для множества Advice-ов
- Возможность комбинирования Pointcut-ов

# Комбинирование Pointcut-ов

Комбинирование Pointcut-ов – это их объединение с помощью логических операторов **&&** **||** **!**

```
@Pointcut("execution(* aop.UniLibrary.get*())")
private void allGetMethodsFromUniLibrary() { }

@Pointcut("execution(* aop.UniLibrary.return*())")
private void allReturnMethodsFromUniLibrary() { }

@Pointcut("allGetMethodsFromUniLibrary() || allReturnMethodsFromUniLibrary()")
private void allGetAndReturnMethodsFromUniLibrary() { }
```

```
@Pointcut("execution( * aop.UniLibrary.*(..) )")
private void allMethodsFromUniLibrary() { }

@Pointcut("execution(* aop.UniLibrary.returnMagazine())")
private void returnMagazineFromUniLibrary() { }

@Pointcut("allMethodsFromUniLibrary() && !returnMagazineFromUniLibrary()")
private void allMethodsExceptReturnMagazineFromUniLibrary() { }
```



# Порядок выполнения Aspect-ов

Если при вызове 1-го метода с бизнес-логикой срабатывают несколько Advice-ов, то нет никакой гарантии в порядке выполнения этих Advice-ов.

Для соблюдения порядка такие Advice-ы нужно распределять по отдельным упорядоченным Aspect-ам.

```
@Component
@Aspect
@Order(2)
public class SecurityAspect {
```

@Order(3) упорядочивает Aspect-ы.

Чем меньше число, тем выше приоритет.

# Join Point

Joint Point – это точка/момент в выполняемой программе когда следует применять Advice. Т.е. это точка переплетения метода с бизнес-логикой и метода со служебным функционалом.

Прописав Joint Point в параметре метода Advice, мы получаем доступ к информации о сигнатуре и параметрах метода с бизнес-логикой.

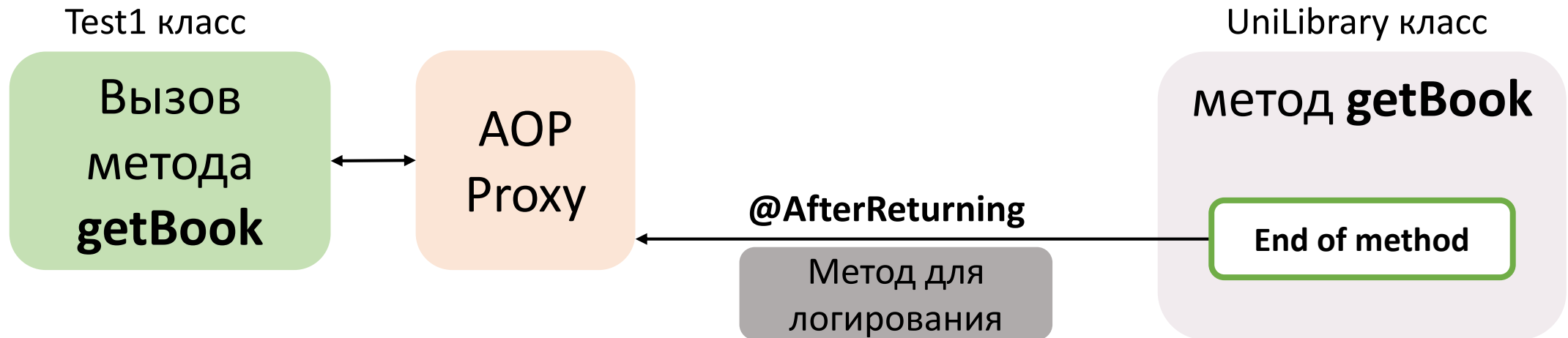
```
@Before("aop.aspects.MyPointcuts.allGetMethods()")  
public void methodAdvice(JoinPoint joinPoint) {  
    MethodSignature methodSignature = (MethodSignature) joinPoint.getSignature();  
    Object[] arguments = joinPoint.getArgs();  
}
```

# Advice Типы

- Before – выполняется до метода с основной логикой
- After returning – выполняется только после нормального окончания метода с основной логикой
- After throwing – выполняется после окончания метода с основной логикой только, если было выброшено исключение
- After/ After finally – выполняется после окончания метода с основной логикой
- Around – выполняется до и после метода с основной логикой

# Advice @AfterReturning

@AfterReturning Advice выполняется только после нормального окончания метода с основной логикой.



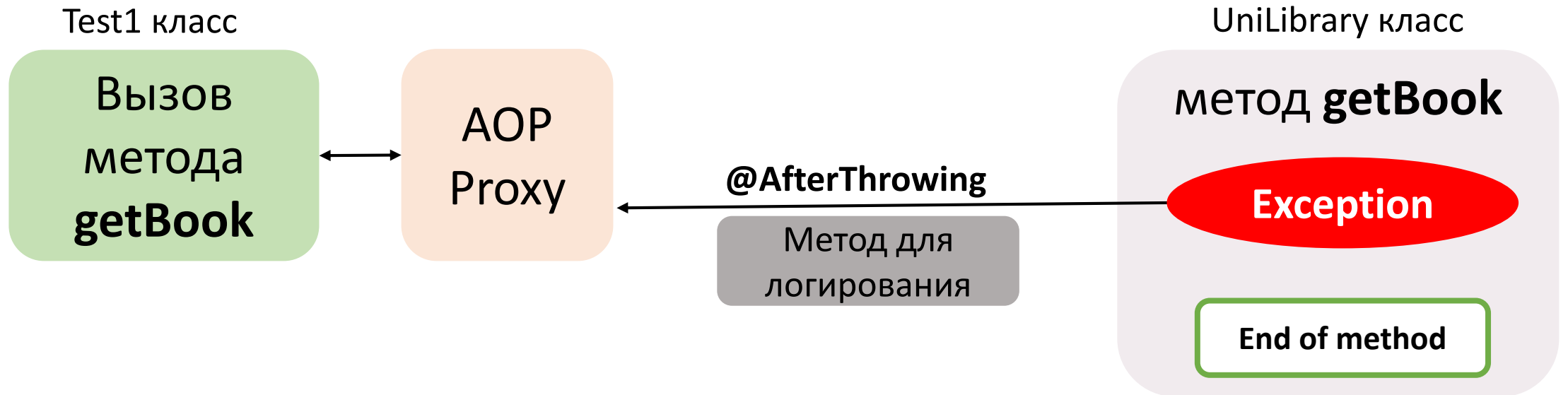
# Advice @AfterReturning

@AfterReturning Advice выполняется только после нормального окончания метода с основной логикой, но до присвоения результата этого метода какой-либо переменной. Поэтому с помощью @AfterReturning Advice возможно изменять возвращаемый результат метода.

```
@AfterReturning(pointcut = "execution(* getStudents())"  
    , returning = "students")  
public void afterReturningGetStudentsLoggingAdvice(JoinPoint joinPoint  
    , List<Student> students) {    //Your code  
}
```

# Advice @AfterThrowing

@AfterThrowing Advice выполняется после окончания работы метода, если в нём было выброшено исключение.



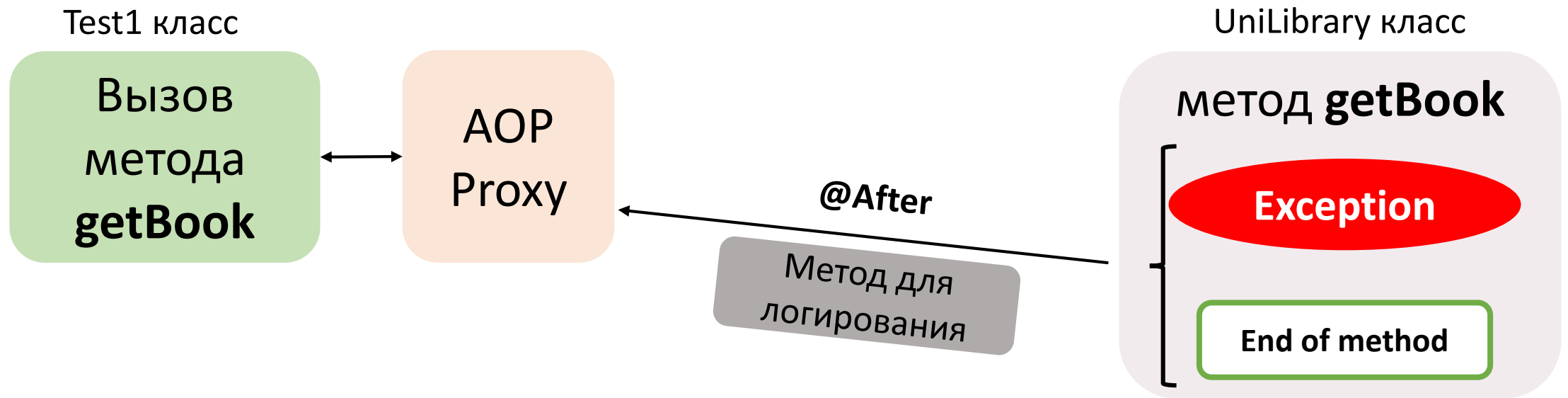
# Advice @AfterThrowing

@AfterThrowing Advice не влияет на протекание программы при выбрасывании исключения. С помощью @AfterThrowing Advice можно получить доступ к исключению, которое выбросилось из метода с основной логикой.

```
@AfterThrowing(pointcut = "execution(* getStudents())"  
    , throwing = "exception")  
public void afterThrowingGetStudentsLoggingAdvice(JoinPoint joinPoint  
    , Throwable exception) {    //Your code  
}
```

# Advice @After

@After Advice выполняется после окончания метода с основной логикой вне зависимости от того, завершается ли метод нормально или выбрасывается исключение.





# Advice @After

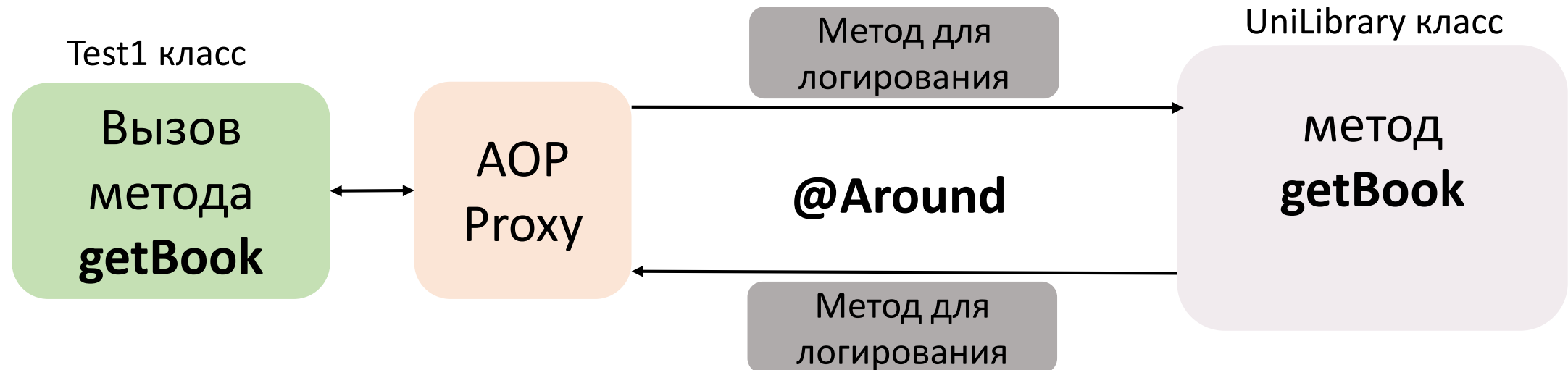
С помощью @After Advice невозможно:

- 1) получить доступ к исключению, которое выбросилось из метода с основной логикой;
- 2) получить доступ к возвращаемому методом результату.

```
@After("execution(* getStudents())")  
public void afterGetStudentsLoggingAdvice(JoinPoint joinPoint)  
{  
    //Your code  
}
```

# Advice @Around

- @Around Advice выполняется до и после метода с основной логикой.



# Advice @Around

С помощью @Around Advice возможно:

- 1) произвести какие-либо действия до работы target метода;
- 2) произвести какие-либо действия после работы target метода;
- 3) получить результат работы target метода/изменить его;
- 4) предпринять какие-либо действия, если из target метода выбрасывается исключение.

```
@Around("execution(public String returnBook())")
public Object aroundReturnBookLoggingAdvice(ProceedingJoinPoint proceedingJoinPoint)
    throws Throwable {
    System.out.println(" aroundReturnBookLoggingAdvice: в библиотеку " +
        "пытаются вернуть книгу");

    Object targetMethodResult = proceedingJoinPoint.proceed();

    System.out.println(" aroundReturnBookLoggingAdvice: в библиотеку " +
        "успешно вернули книгу");
    return targetMethodResult;
}
```

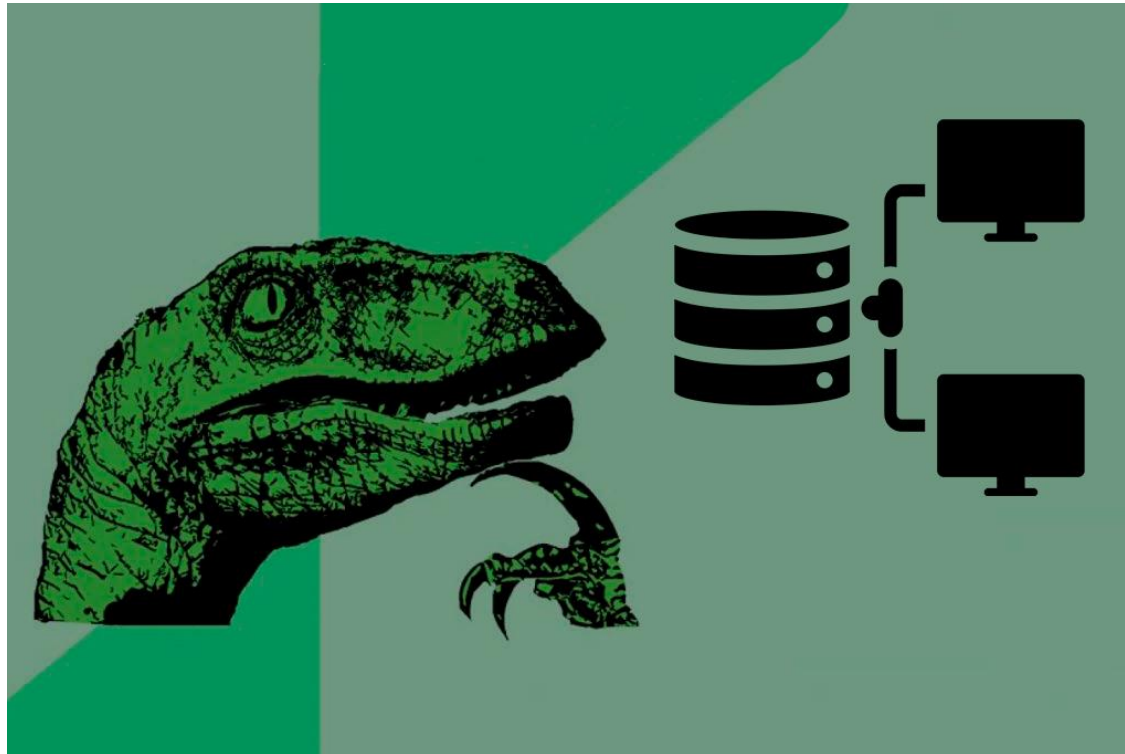
# Advice @Around. Работа с исключениями

Используя @Around Advice возможно предпринять следующие действия, если из target метода выбрасывается исключение:

- Ничего не делать
- Обработать исключение
- Пробрасывать исключение дальше

# Hibernate

SQL для начинающих: с нуля до сертификата Oracle



# Hibernate

Для изучения Hibernate необходимы  
минимальные знания SQL

- SELECT
- INSERT
- UPDATE
- DELETE

# Hibernate

Hibernate – это framework, который используется для сохранения, получения, изменения и удаления Java объектов из Базы Данных.



# Hibernate

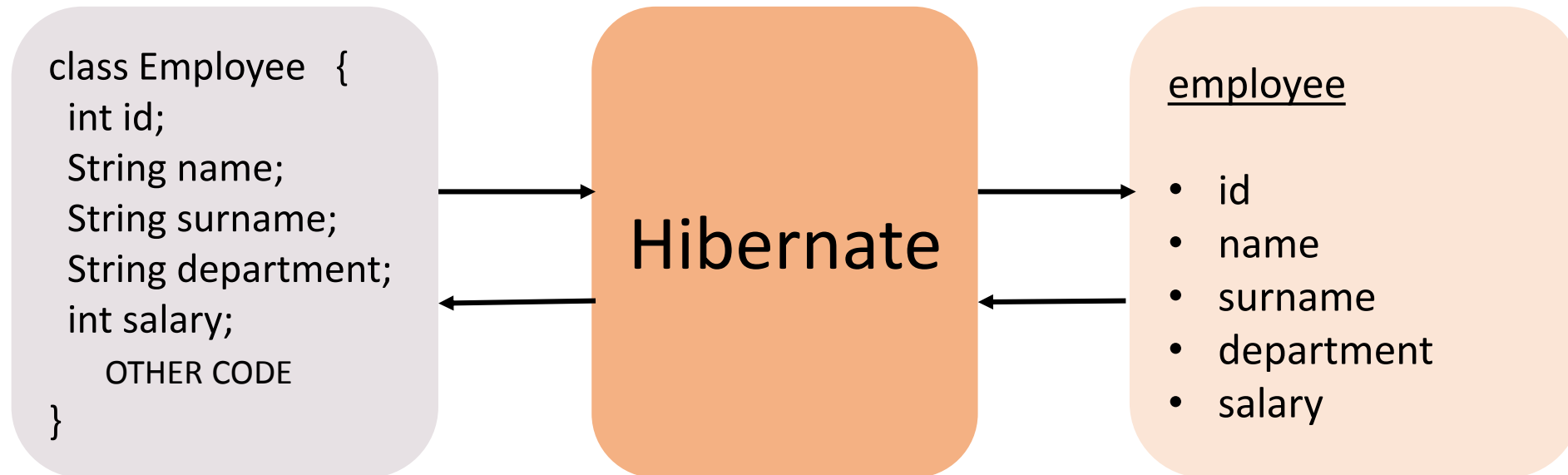
## Плюсы Hibernate:

- Предоставляет технологию ORM
- Регулирует SQL запросы
- Уменьшает количество кода для написания



# Hibernate

ORM (Object-to-Relational Mapping) – это преобразование объекта в строку в таблице и обратное преобразование



# Hibernate

Java application  
save



Hibernate

- Сбор данных из полей объекта;
- Написание INSERT выражения для добавления новой строки в таблицу с собранными данными.

Java application  
get

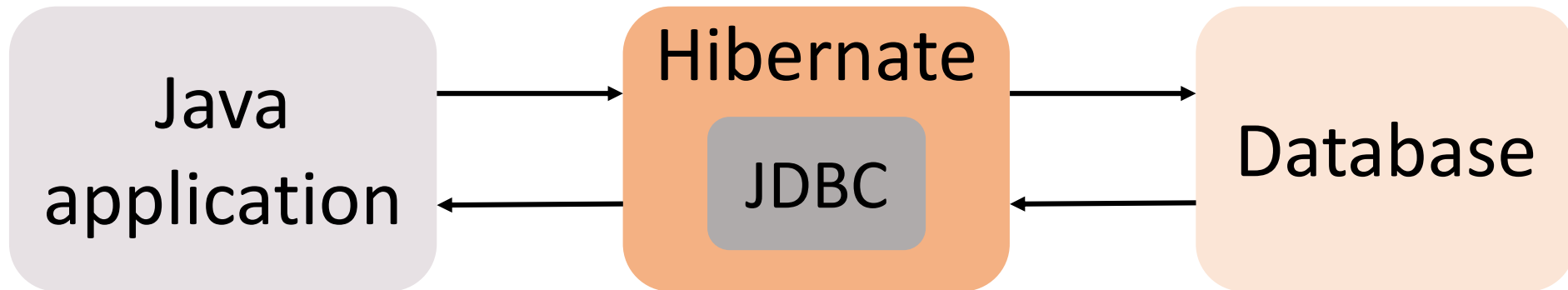


Hibernate

- Написание SELECT выражения для получения необходимых данных;
- Создание объекта Java класса и присвоение его полям значений, полученных из базы.

# Hibernate

Hibernate использует JDBC для работы с базой данных.



# Hibernate

## CRUD

- **CREATE** - команда INSERT
- **READ** - команда SELECT
- **UPDATE** - команда UPDATE
- **DELETE** - команда DELETE

# Hibernate

## Подготовка окружения для работы с Hibernate:

- Установка MySQL

- Root pwd: springcourse
- Connection: my\_connection
- User: bestuser
- Pwd: bestuser
- DB: my\_db

- Подключение Java приложения к базе

- Подключение Hibernate
- MySQL JDBC Driver
- hibernate.cfg.xml

# Hibernate

Конфигурировать связь между классом и таблицей можно 2-мя способами:

- С помощью XML файла
- С помощью Java аннотаций

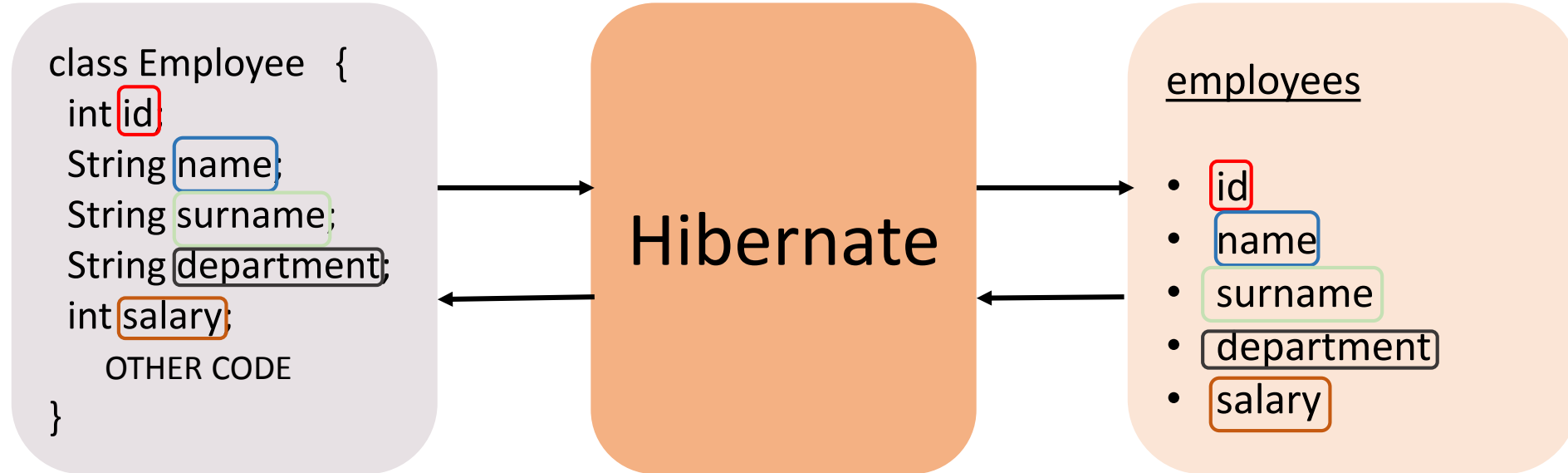
# Hibernate

Entity класс – это Java класс, который отображает информацию определённой таблицы в Базе Данных

Entity класс – это POJO класс, в котором мы используем определённые Hibernate аннотации для связи класса с таблицей из базы

POJO (Plain Old Java Object) – класс, удовлетворяющий ряду условий:  
private поля, getter-ы и setter-ы, конструктор без аргументов и т.д.

# Hibernate





# Hibernate

Аннотация `@Entity` говорит о том, что данный класс будет иметь отображение в базе данных

Аннотация `@Table` говорит о том, к какой именно таблице мы привязываем класс

Аннотация `@Column` говорит о том, к какому именно столбцу из таблицы мы привязываем поле класса

Аннотация `@Id` говорит о том, что в таблице, столбец связанный с данным полем является Primary Key

# Hibernate

JPA (Java Persistence API) – это стандартная спецификация, которая описывает систему для управления сохранением Java объектов в таблицы базы данных

Hibernate – самая популярная реализация спецификации JPA

Таким образом JPA описывает правила, а Hibernate реализует их.

# SessionFactory

SessionFactory – фабрика по производству сессий.

- SessionFactory читает файл hibernate.cfg.xml  
После чего SessionFactory знает, как должны создаваться сессии.
- В Java приложении достаточно создать объект SessionFactory 1 раз и затем можно его переиспользовать.

# Session

Session – это обёртка вокруг подключения к базе с помощью JDBC.

- Session мы получаем с помощью SessionFactory
- Session – это основа для работы с Базой Данных. Именно с помощью Session мы будем добавлять, получать и делать другие операции с Java Объектами в Базе Данных.
- Жизненный цикл Session обычно не велик. Мы получаем Session, делаем с помощью неё определённые операции и она становится не нужной

# Hibernate

## Добавление объекта в таблицу

```
public class TestEmployee {  
    public static void main(String[] args) {  
        SessionFactory factory = new Configuration()  
            .configure("hibernate.cfg.xml")  
            .addAnnotatedClass(Employee.class)  
            .buildSessionFactory();  
  
        try{  
            Employee emp = new Employee( name: "Zaur2", surname: "Tregulov"  
                , department: "IT", salary: 500);  
            Session session = factory.getCurrentSession();  
            session.beginTransaction();  
            session.save(emp);  
            session.getTransaction().commit();  
        }  
        finally {  
            factory.close();  
        }  
    }  
}
```

# Primary Key

Столбец с Primary Key содержит уникальное значение и не может быть null.

Аннотация `@GeneratedValue` описывает стратегию по генерации значений для столбца с Primary Key.

**GenerationType.AUTO** – дефолтный тип. Выбор стратегии будет зависеть от типа базы, с которой мы работаем.

**GenerationType.IDENTITY** полагается на автоматическое увеличение столбца по правилам, прописанным в БД-х.

**GenerationType.SEQUENCE** полагается на работу Sequence, созданного в БД-х.

**GenerationType.TABLE** полагается на значение столбца таблицы БД-х. Цель такой таблицы – поддержка уникальности значений.

# @GeneratedValue

Пример использования @GeneratedValue:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column
private int id;
```

# Получение объекта из базы по ID

```
int id = emp.getId();  
Session session = factory.getCurrentSession();  
session.beginTransaction();  
Employee myEmp = session.get(Employee.class, id);  
session.getTransaction().commit();  
System.out.println(myEmp);
```



# Получение объектов из базы

Для получения объектов из базы используется HQL (Hibernate Query Language). HQL очень схож с SQL.

```
Session session = factory.getCurrentSession();
session.beginTransaction();

List<Employee> emps = session.createQuery("from Employee")
    .getResultList();
for(Employee e: emps){
    System.out.println(e);
}

session.getTransaction().commit();
```

# Получение объектов из базы

```
Session session = factory.getCurrentSession();
session.beginTransaction();

List<Employee> emps = session.createQuery("from Employee " +
    "where firstName = 'Aleksandr'")
    .getResultList();
for(Employee e: emps){
    System.out.println(e);
}

session.getTransaction().commit();
```

```
Session session = factory.getCurrentSession();
session.beginTransaction();

List<Employee> emps = session.createQuery("from Employee " +
    "where firstName = 'Aleksandr' and salary > 650")
    .getResultList();
for(Employee e: emps){
    System.out.println(e);
}

session.getTransaction().commit();
```

# Update объектов

```
Session session = factory.getCurrentSession();  
session.beginTransaction();  
Employee emp1 = session.get(Employee.class, serializable: 1);  
emp1.setSalary(100);  
session.getTransaction().commit();
```

```
Session session = factory.getCurrentSession();  
session.beginTransaction();  
session.createQuery(s: "update Employee set salary = 1000 " +  
    "where firstName='Aleksandr'").executeUpdate();  
session.getTransaction().commit();
```

# Delete объектов

```
Session session = factory.getCurrentSession();  
session.beginTransaction();  
Employee emp1 = session.get(Employee.class, serializable: 1);  
session.delete(emp1);  
session.getTransaction().commit();
```

```
Session session = factory.getCurrentSession();  
session.beginTransaction();  
session.createQuery(s: "delete Employee " +  
    "where firstName = 'Aleksandr'").executeUpdate();  
session.getTransaction().commit();
```

# Типы отношений таблиц

One-to-One (отношение Один-к-Одному)

School

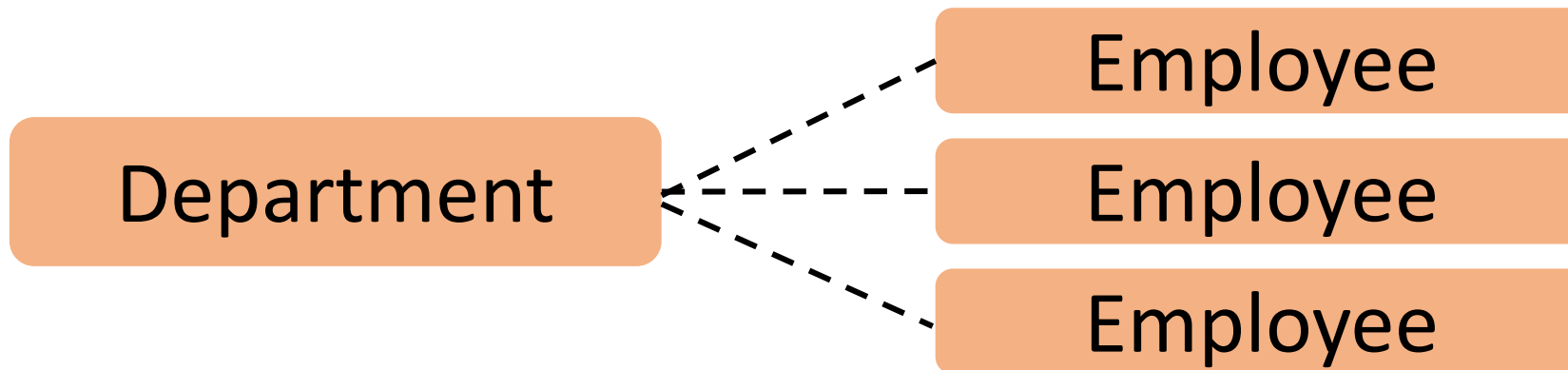
Director

Employee

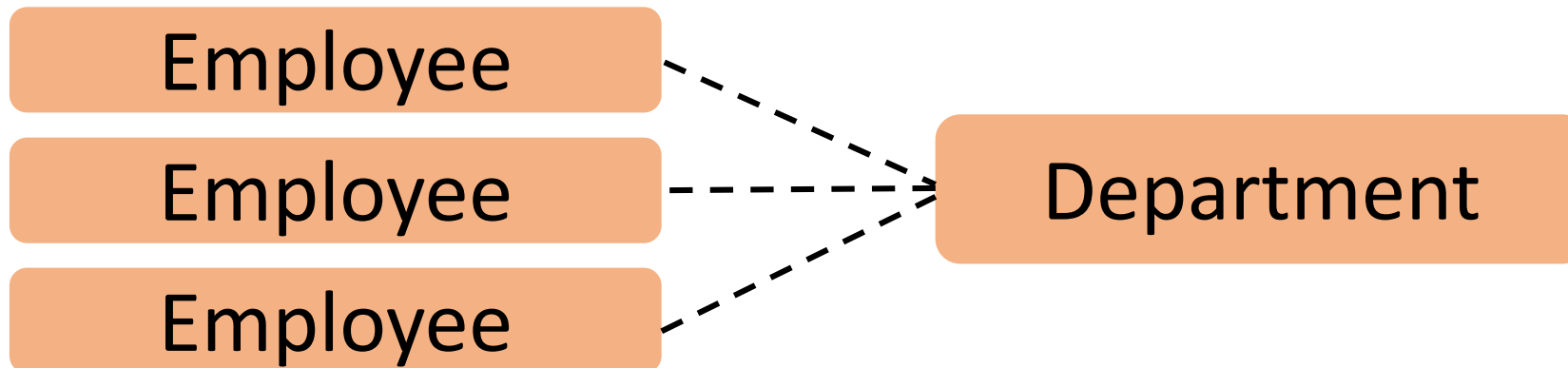
Details

# Типы отношений таблиц

One-to-Many (отношение Один-ко-Многим)

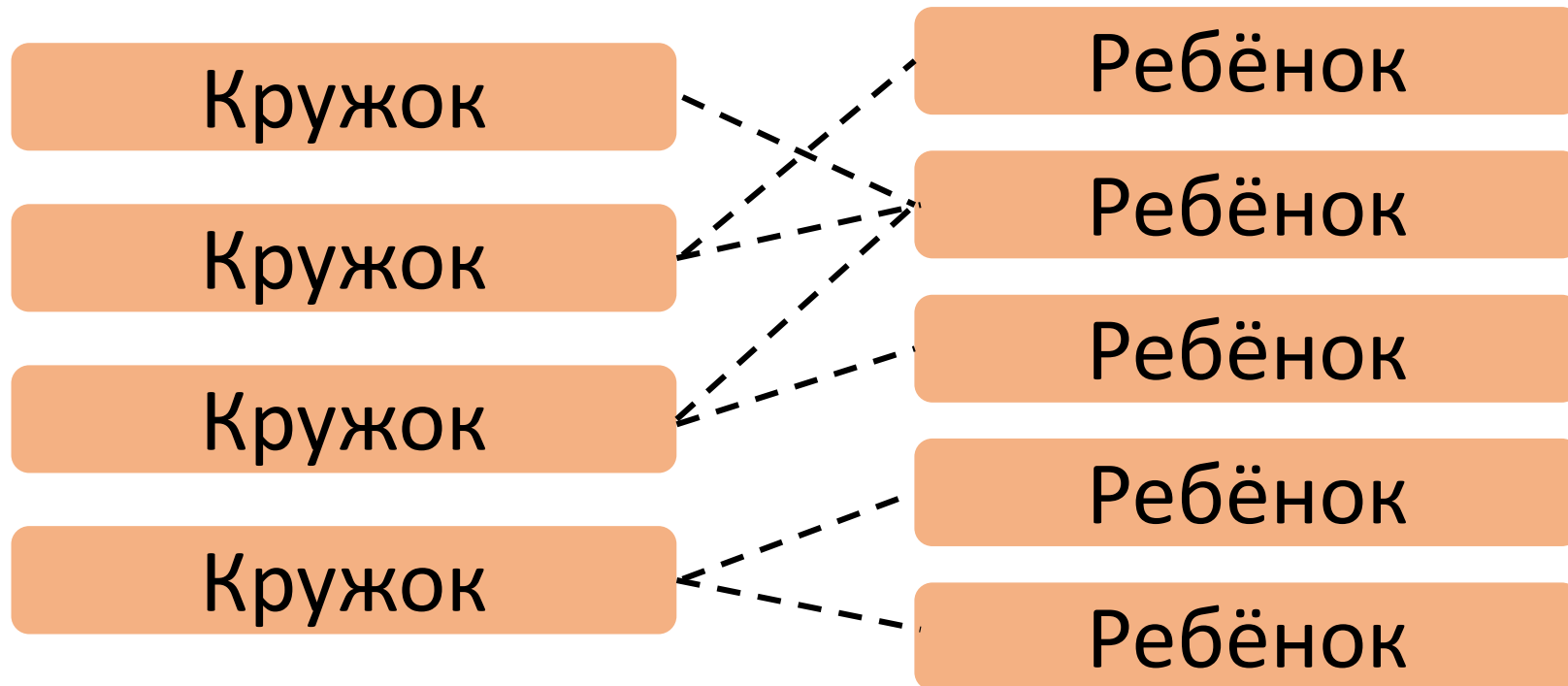


Many-to-One (отношение Многие-к-Одному)



# Типы отношений таблиц

Many-to-Many (отношение Многие-ко-Многим)



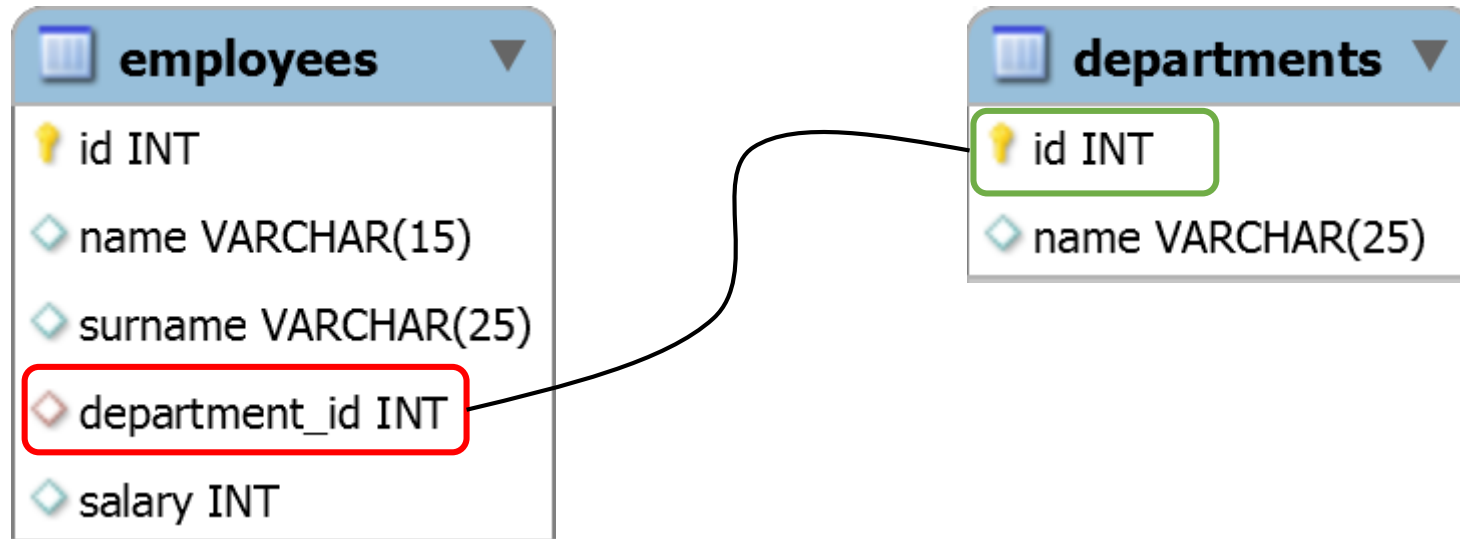
# Foreign Key

Foreign Key – это внешний ключ.

- FK используется для создания связи между двумя таблицами
- Обычно FK это столбец, который ссылается на PK столбец другой таблицы
- FK столбец не может содержать информацию, которой нет в столбце, на который он ссылается



# Foreign Key



id	name	surname	department_id	salary
1	Zaur	Tregulov	1	500
2	Aleksandr	Smirnov	2	600
3	Elena	Ivanova	2	700
4	Oleg	Petrov	3	800

id	name
1	IT
2	Sales

# One-to-One

School

Director

Employee

Details

# Uni и Bi-directional associations

```
class Parent {  
    Child child;  
}  
  
class Child {  
  
}
```

Uni-directional - это  
отношения, когда одна  
сторона о них не знает

```
class Parent {  
    Child child;  
}  
  
class Child {  
    Parent parent;  
}
```

Bi-directional - это  
отношения, когда обе  
стороны знают друг о друге

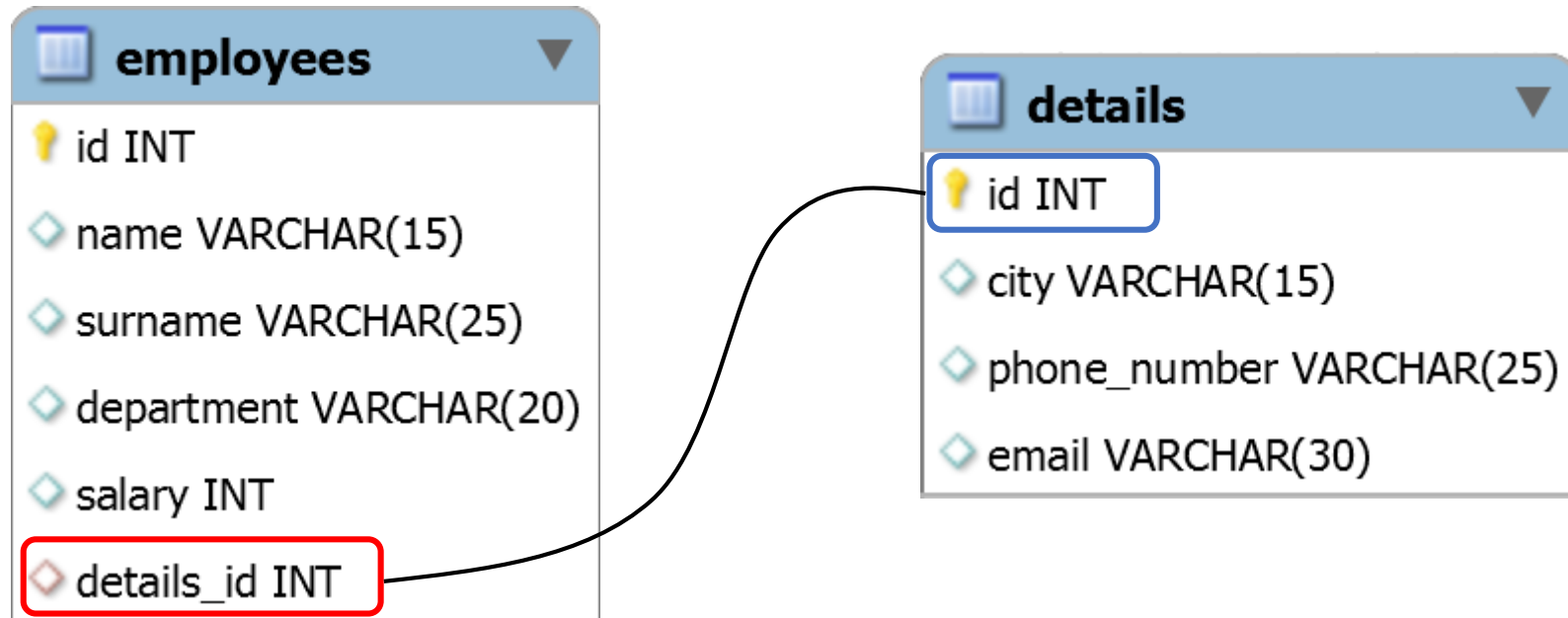
# One-to-One

Employee



Details

# Foreign Key



# One-to-One

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "details_id")
private Detail empDetail;
```

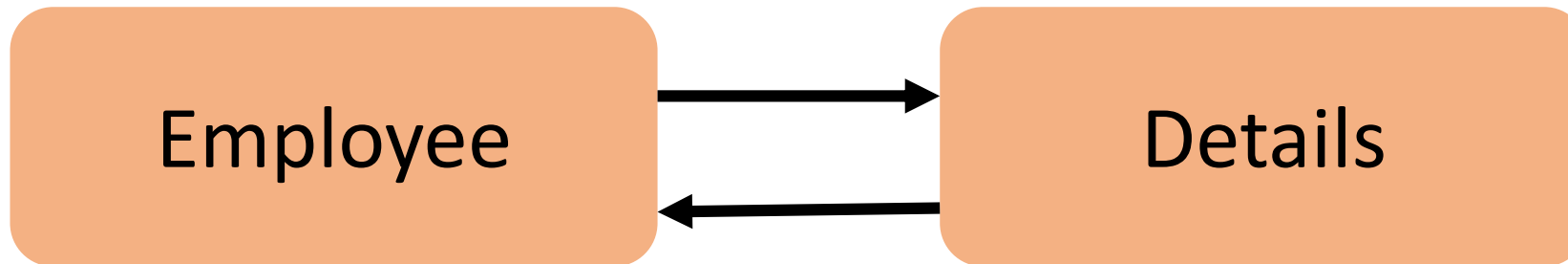
Аннотация @OneToOne указывает на тип отношений между объектами

Аннотация @JoinColumn указывает на столбец, который осуществляет связь с другим объектом

Cascade операций – это выполнение операции не только для Entity, на котором операция вызывается, но и на связанных с ним Entity

# One-to-One

Bi-directional - это отношения, когда обе стороны знают друг о друге

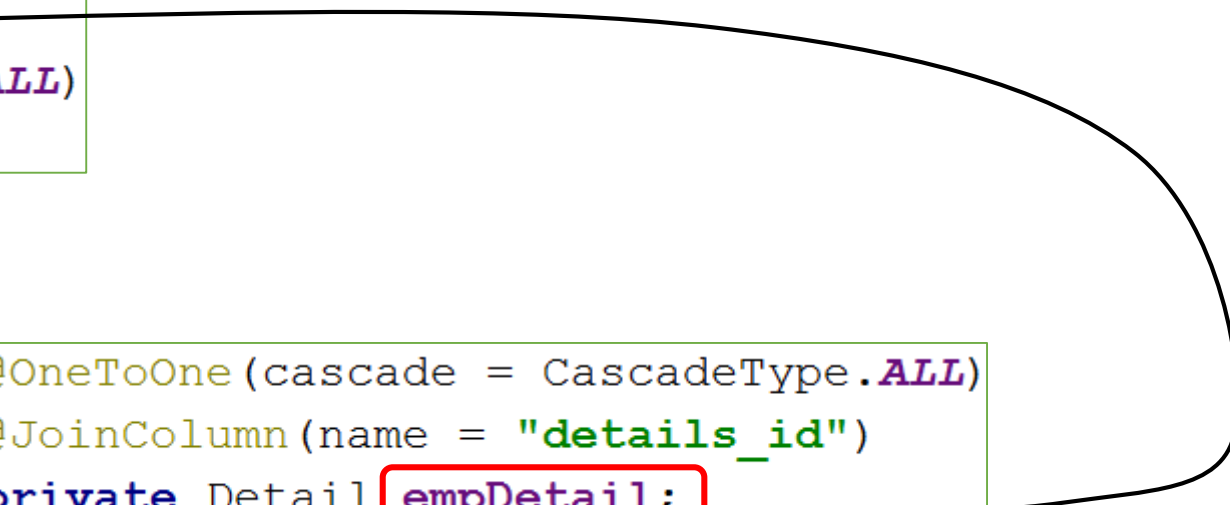


# One-to-One

В Bi-directional отношениях с помощью аннотации `@OneToOne` и `mappedBy` мы показываем Hibernate, где нужно искать связь между классами

```
@OneToOne(mappedBy = "empDetail"  
            , cascade = CascadeType.ALL)  
private Employee employee;
```

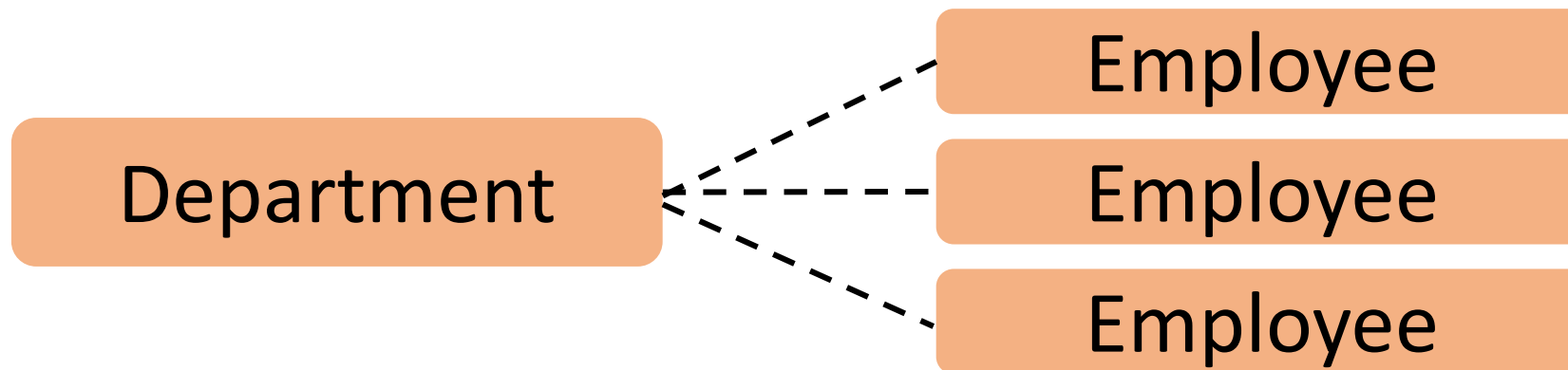
```
@OneToOne(cascade = CascadeType.ALL)  
@JoinColumn(name = "details_id")  
private Detail empDetail;
```



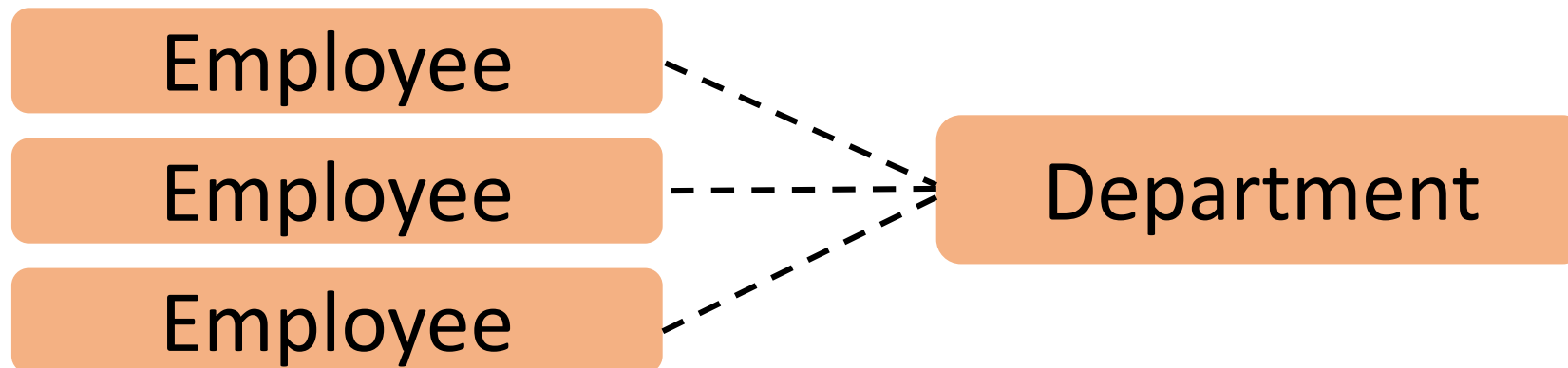


# Типы отношений таблиц

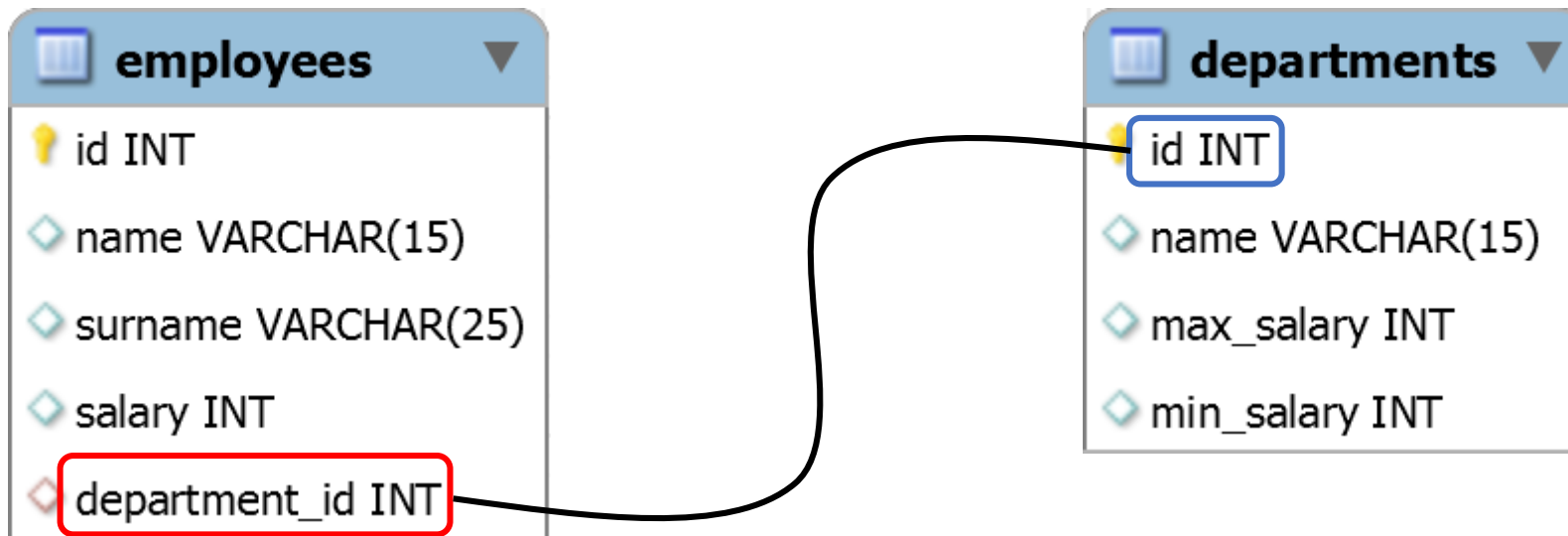
One-to-Many (отношение Один-ко-Многим)



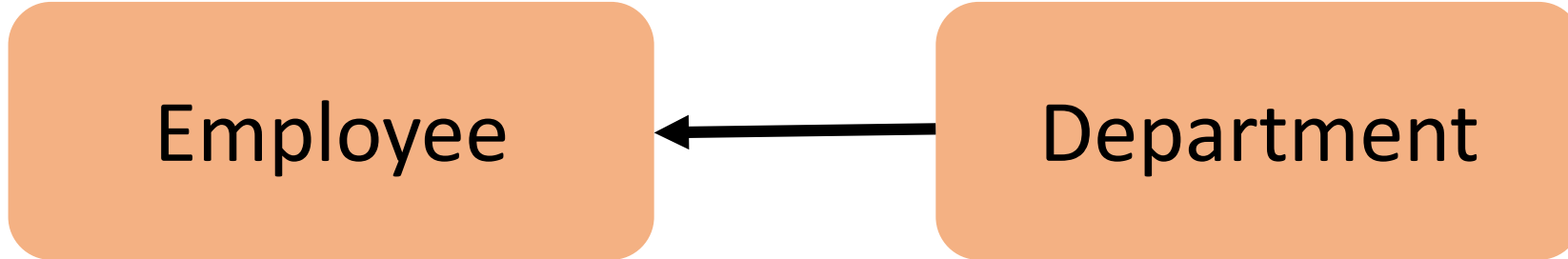
Many-to-One (отношение Многие-к-Одному)



# Foreign Key



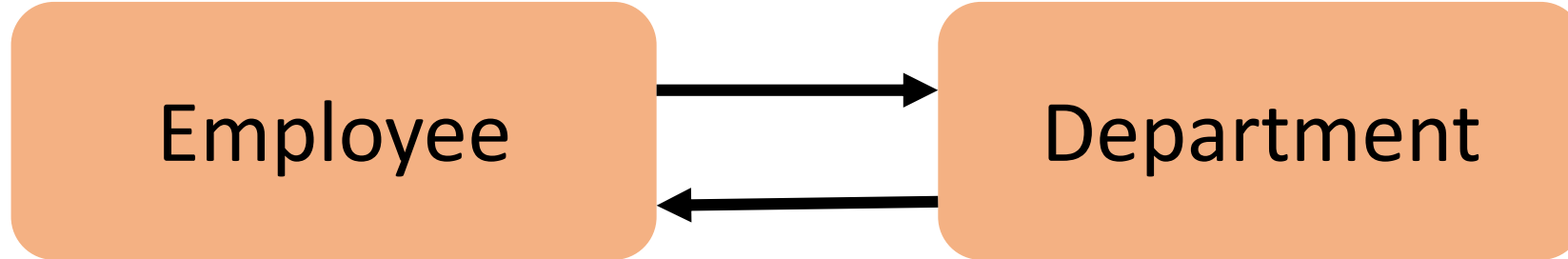
# Uni-directional One-to-Many



```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "department_id")
private List<Employee> emps;
```

При использовании связи One-to-Many в аннотации `@JoinColumn` `name` будет ссылаться на Foreign Key не из source, а из target таблицы.

# Bi-directional One-to-Many



```
@OneToMany(mappedBy = "department"  
            , cascade = CascadeType.ALL)  
private List<Employee> emps;
```

```
@ManyToOne(cascade = CascadeType.ALL)  
@JoinColumn(name = "department_id")  
private Department department;
```

# Loading types

## Типы загрузки данных:

- Eager (нетерпеливая) загрузка – при её использовании связанные сущности загружаются сразу вместе с загрузкой основной сущности
- Lazy (ленивая) загрузка – при её использовании связанные сущности НЕ загружаются сразу вместе с загрузкой основной сущности. Связанные сущности загрузятся только при первом обращении к ним

# Loading types

В большинстве случаев при большом количестве связанных сущностей целесообразнее использовать Lazy loading по следующим причинам:

- Lazy загрузка имеет лучший performance по сравнению с Eager загрузкой
- Иногда при загрузке основной сущности, нам просто не нужны связанные с ней сущности. Поэтому их загрузка – это лишняя работа.

# Fetch type (Тип выборки)

Типы выборки по умолчанию

Тип связи	Тип загрузки
One-to-One	<b>Eager</b>
One-to-Many	<b>Lazy</b>
Many-to-One	<b>Eager</b>
Many-to-Many	<b>Lazy</b>

# Fetch type (Тип выборки)

```
@OneToMany(mappedBy = "department"  
            , cascade = CascadeType.ALL  
            , fetch = FetchType.EAGER)  
private List<Employee> emps;
```

**EAGER** loading

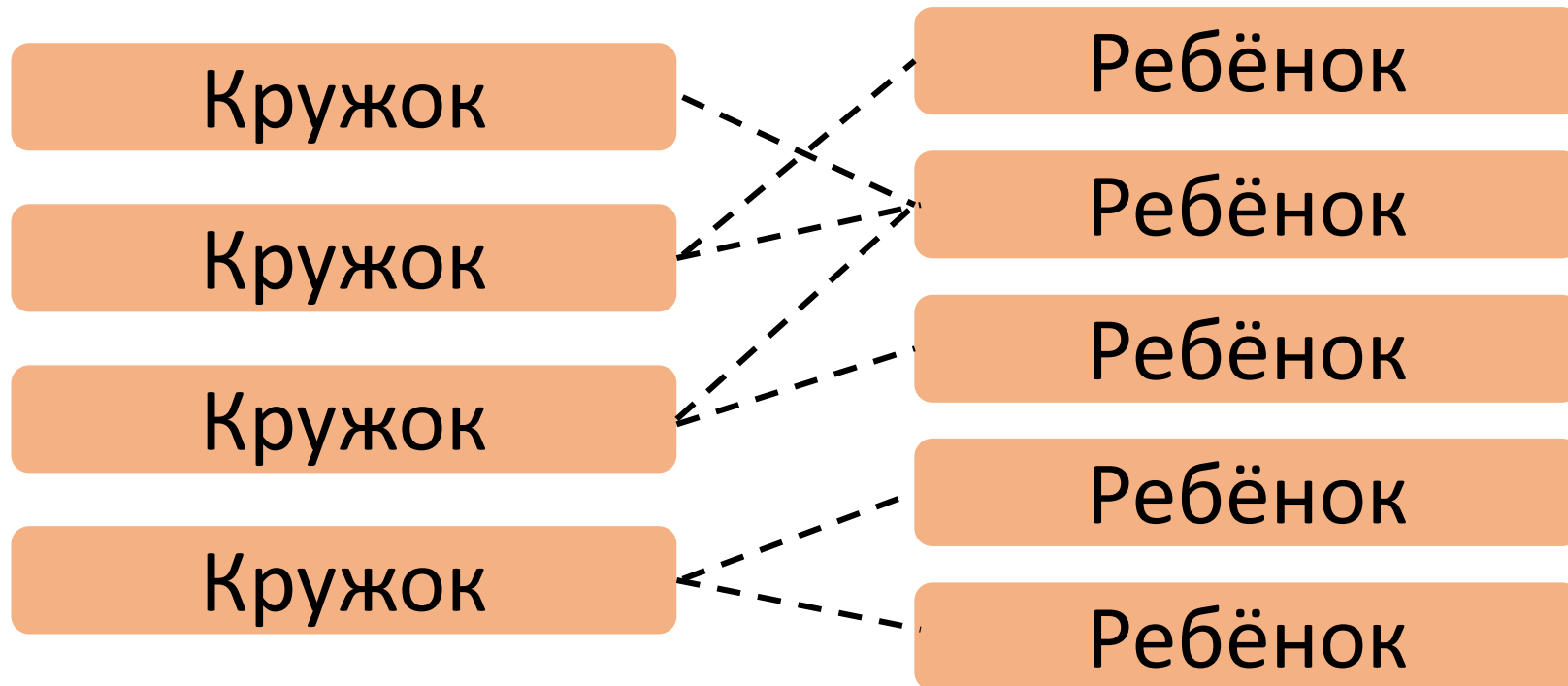
**LAZY** loading

```
@OneToMany(mappedBy = "department"  
            , cascade = CascadeType.ALL  
            , fetch = FetchType.LAZY)  
private List<Employee> emps;
```



# Типы отношений таблиц

Many-to-Many (отношение Многие-ко-Многим)

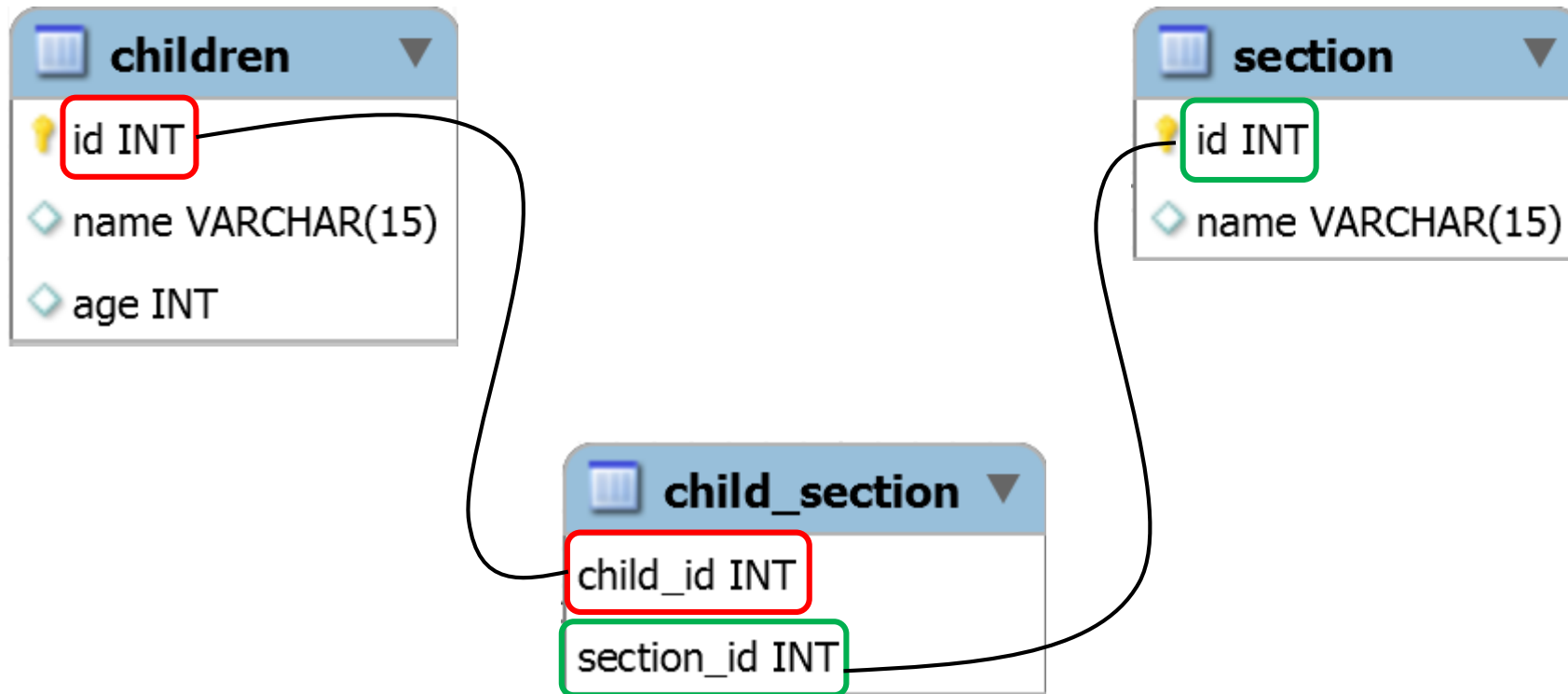


# Join Table

Join Table – это таблица, которая отображает связь между строками 2-х других таблиц

Столбцы Join Table – это Foreign Key, которые ссылаются на Primary Key связываемых таблиц

# Join Table



# Join Table

В аннотации @JoinTable:

- Мы прописываем название таблицы, которая выполняет роль Join Table
- В joinColumns мы указываем столбец таблицы Join Table, который ссылается на Primary Key source таблицы
- В inverseJoinColumns мы указываем столбец таблицы Join Table, который ссылается на Primary Key target таблицы

# Join Table

## class Section

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
    name = "child_section"
    , joinColumns = @JoinColumn(name = "section_id")
    , inverseJoinColumns = @JoinColumn(name = "child_id"))
private List<Child> children;
```

## class Child

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
    name = "child_section"
    , joinColumns = @JoinColumn(name = "child_id")
    , inverseJoinColumns = @JoinColumn(name = "section_id"))
private List<Section> sections;
```

# Тренажёрный зал 😊

1. Приседания с максимальным весом
2. Приседания с минимальным весом
3. Подтягивания
4. Упражнения с гантелями

Ошибки новичка

1. Разминка
2. Беговая дорожка
3. Приседания с минимальным весом
4. Приседания с более тяжёлым весом

Советы профи

# MVC (Model View Controller)

Model – компонент, отвечающий за данные приложения и за работу с этими данными

View (представление) – компонент, отвечающий за взаимодействие с пользователем. View отображает данные и определяет внешний вид приложения.

Controller – компонент, отвечающий за связь между View и Model. Именно здесь сосредоточена логика работы приложения (бизнес логика)

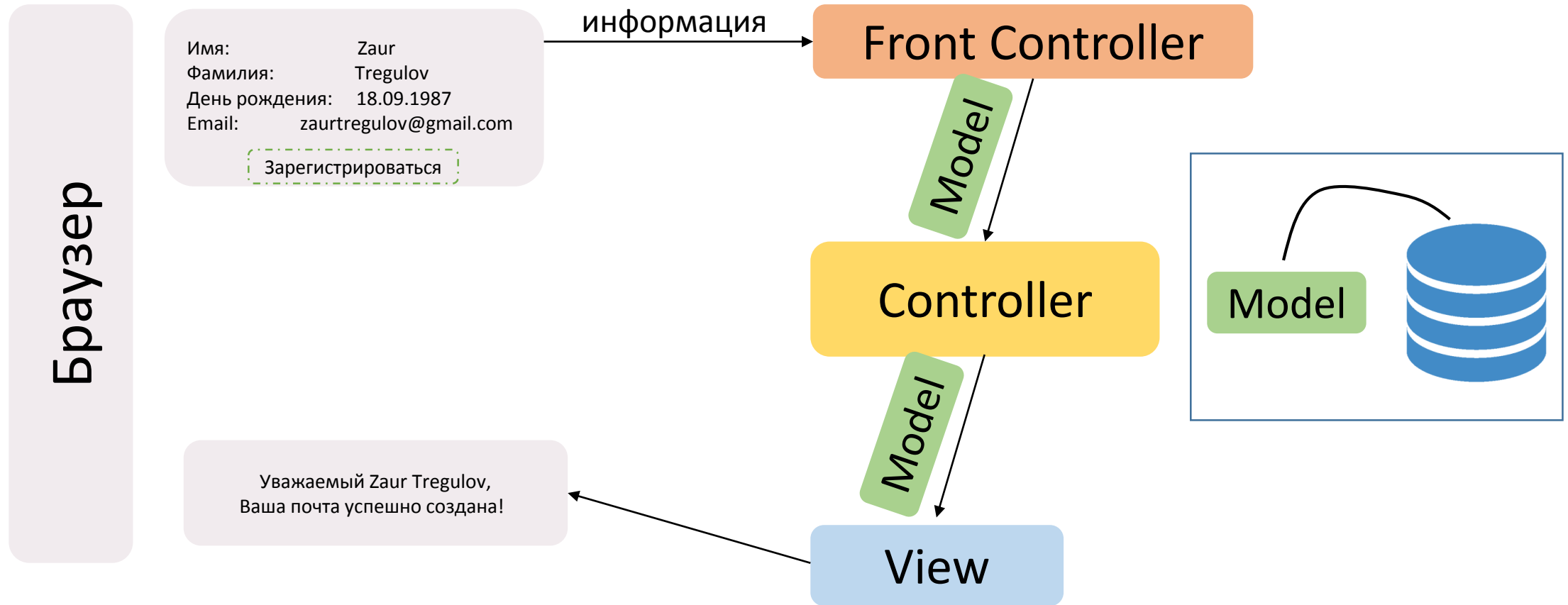
# Spring MVC

Spring MVC – это фреймворк для создания web приложений на Java, в основе которого лежит шаблон проектирования MVC

Естественно, в Spring MVC мы можем использовать весь основной функционал Spring: IoC, DI



# Пример Spring MVC



# Spring MVC

Front Controller также известен под именем DispatcherServlet. Он является частью Spring.

Controller – центр управления, мозг Spring MVC приложения.

Model – контейнер для хранения данных.

# Spring MVC

View – web страница, которую можно создать с помощью html, jsp, Thymeleaf и т.д.  
Часто при отображении View использует данные из Model.

JSP – Java Server Page. Представляет собой html код с небольшим добавлением кода Java

JSTL - Java Server Pages Standart Tag Library. Это расширение спецификации JSP.

# Spring MVC

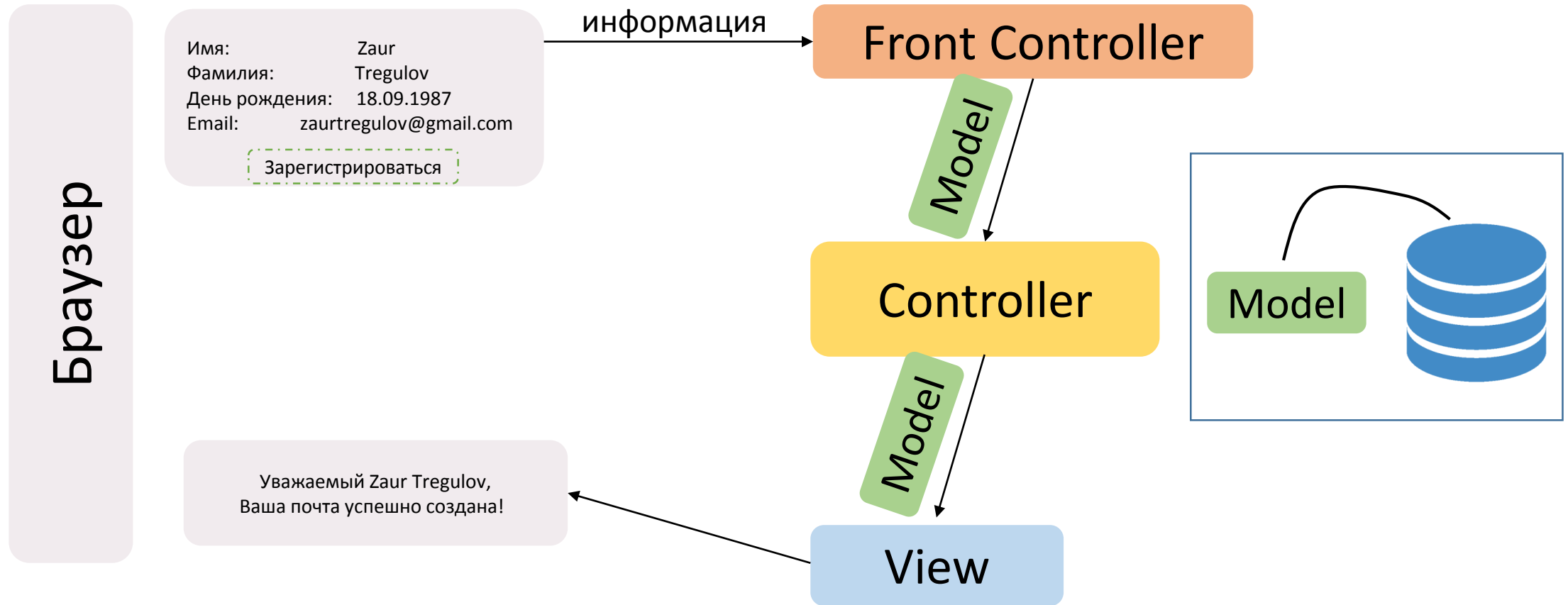
Из чего будет состоять Spring MVC приложение:

- Конфигурация Spring
- Описание Spring бинов
- Web страницы

# Конфигурация Spring MVC

1. Создаём maven проект, используя maven-archetype-webapp
2. Добавляем зависимости в pom файл
3. Скачиваем Tomcat и связываем его со средой разработки
4. Добавляем папки/пакеты в иерархию классов
5. Конфигурируем web.xml
6. Конфигурируем applicationContext.xml

# Пример Spring MVC

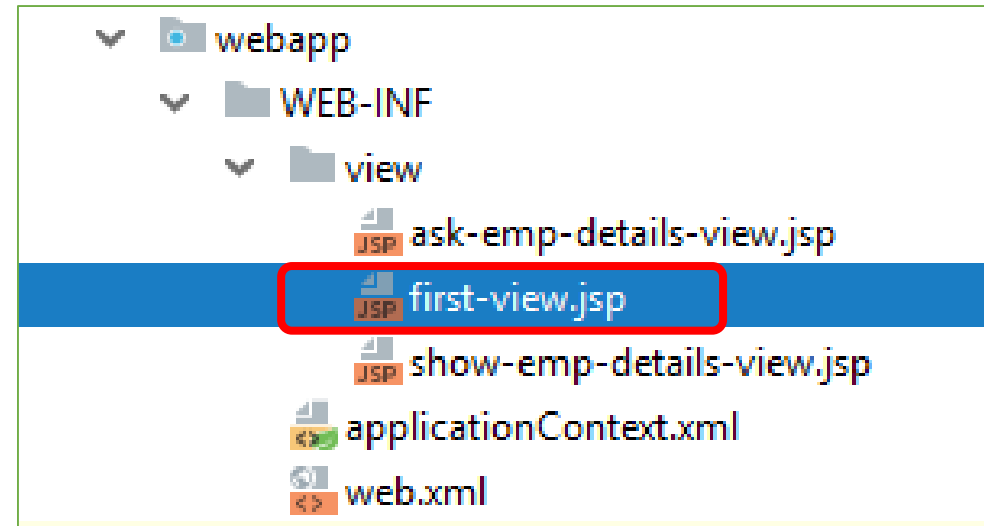


# Controller

```
@Controller
public class MyController {

    @RequestMapping("/")
    public String showFirstView() {
        return "first-view";
    }

}
```



@Controller – это специализированный @Component

@RequestMapping связывает URL (адрес) с методом контроллера.

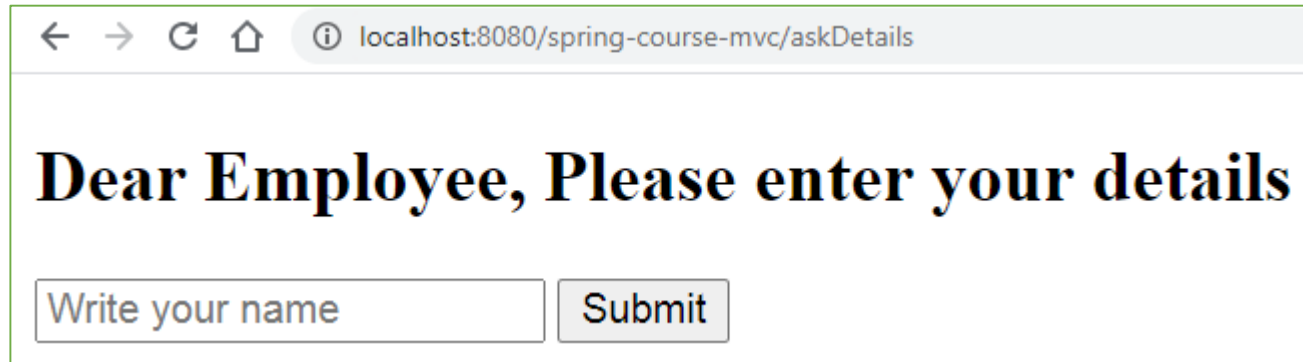
# Controller

Методы в Controller могут:

- Называться как угодно
- Иметь параметры
- Возвращать не только String

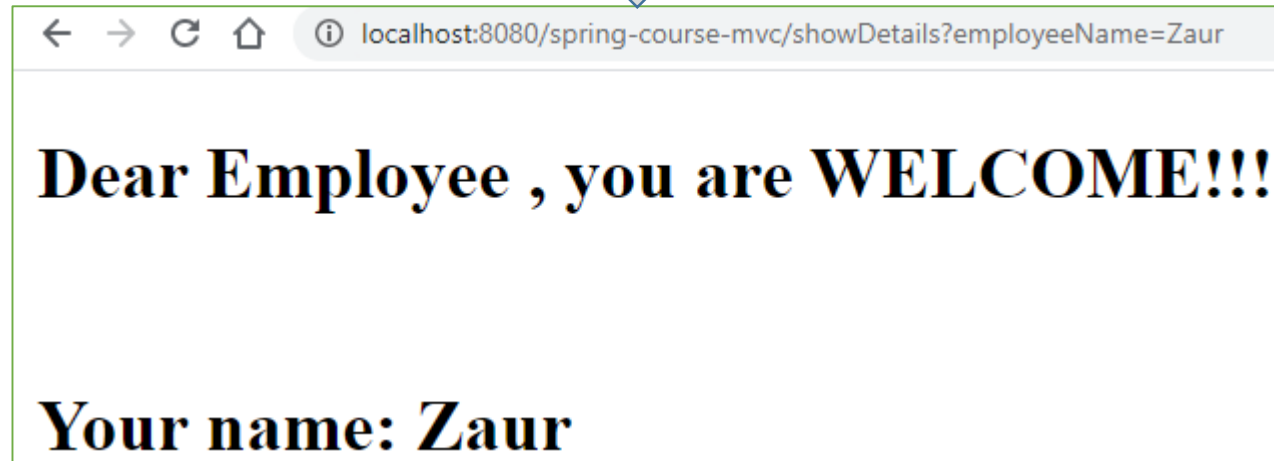


# Чтение данных из формы



← → ↻ 🏠 ⓘ localhost:8080/spring-course-mvc/askDetails

**Dear Employee, Please enter your details**



← → ↻ 🏠 ⓘ localhost:8080/spring-course-mvc/showDetails?employeeName=Zaur

**Dear Employee , you are WELCOME!!!**

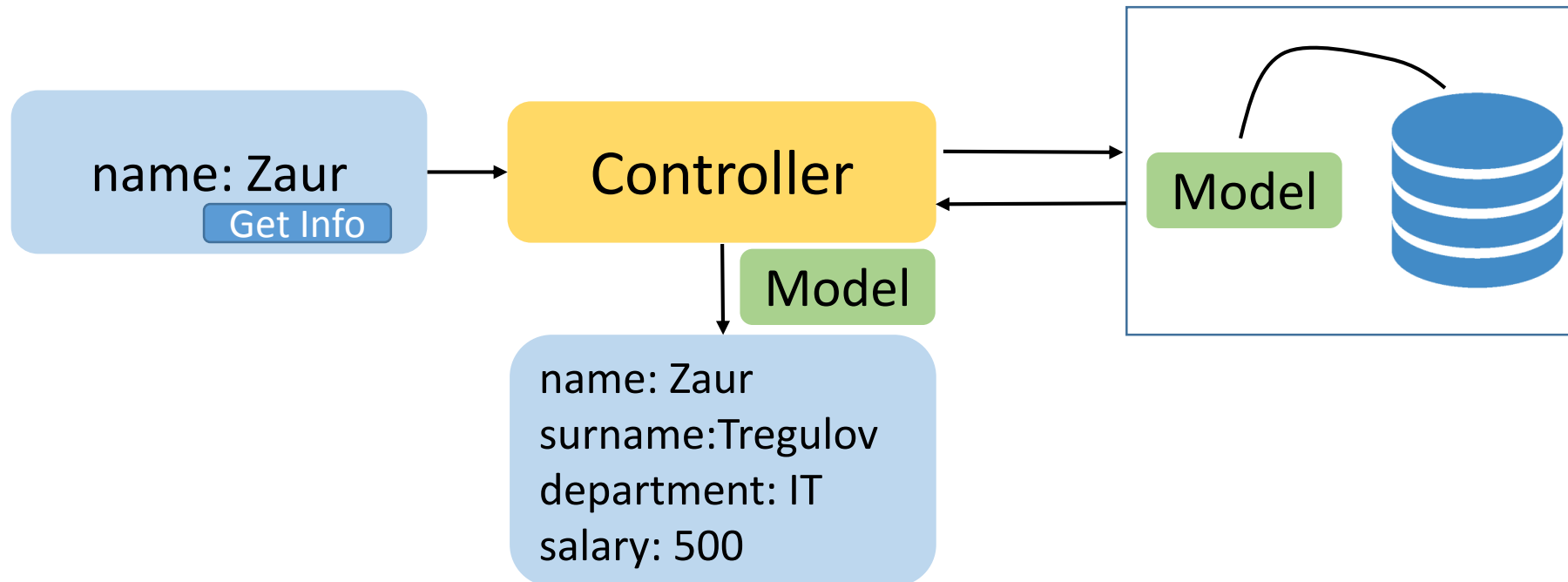
**Your name: Zaur**

# Чтение данных из формы



# Model

Model – контейнер для хранения данных.  
Находясь в Controller, мы можем добавлять данные  
в Model. И затем эти данные использовать во View.



# Model

```
@RequestMapping("/showDetails")
public String showEmployeeDetails(HttpServletRequest request
    , Model model) {

    String empName = request.getParameter(s: "employeeName");
    empName = "Mr. " + empName;
    model.addAttribute(s: "nameAttribute", empName);

    return "show-emp-details-view";
}
```

Your name: `${nameAttribute}`

Обращение к  
атрибуту из View

Имя атрибута

Значение атрибута

# @RequestParam

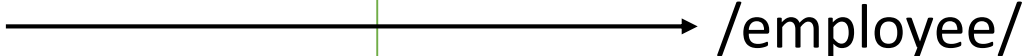
```
@RequestMapping("/showDetails")  
public String showEmployeeDetails(  
    @RequestParam("employeeName") String empName  
    , Model model) {
```

При работе с формами, аннотация @RequestParam позволяет нам связывать поле формы с параметром метода из Controller-a

# @RequestMapping для Controller


```
@Controller  
@RequestMapping("/employee")  
public class MyController {
```

```
    @RequestMapping("/")  
    public String showFirstView() {  
        return "first-view";  
    }
```



\_\_\_\_\_→ /employee/

```
    @RequestMapping("/askDetails")  
    public String askEmployeeDetails() {  
        return "ask-emp-details-view";  
    }
```



\_\_\_\_\_→ /employee/askDetails

@RequestMapping, указанный для Controller-а  
связывает URL (адрес) со всеми его методами

# Ambiguous mapping

```
@Controller
public class MyController {

    @RequestMapping("/askDetails")
    public String askEmployeeDetails() {
        return "ask-emp-details-view";
    }

}
```

```
@Controller
public class TestController {

    @RequestMapping("/askDetails")
    public String testMethod() {
        return "test-view";
    }

}
```

Caused by: java.lang.IllegalStateException: Ambiguous mapping.

# Формы Spring MVC

**Dear Employee, Please enter your details**

Name

Surname

Salary

**Dear Employee , you are WELCOME!!!**

**Your name: Zaur**

**Your surname: Tregulov**

**Your salary: 500**

Model

Employee

- name
- surname
- salary

```
model.addAttribute(s: "employee", new Employee());
```



# Формы Spring MVC

**form:form** – основная форма, содержащая в себе другие формы. Другими словами, форма-контейнер

**form:input** – форма, предназначенная для текста.  
Используется всего лишь одна строка

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

```
<form:form action="showDetails" modelAttribute="employee">
```

```
    Name <form:input path="name"/>
```

```
    <input type="submit" value="OK"/>
```

```
</form:form>
```

# Формы Spring MVC

Your name: `${employee.name}`

Во View обращение к значению идёт по имени атрибута и его полю

```
@RequestMapping("/showDetails")
public String showEmployeeDetails(
    @ModelAttribute("employee") Employee emp) {

    String name = emp.getName();
    emp.setName("Mr " + name);

    return "show-emp-details-view";
}
```

При работе с формами, аннотация `@ModelAttribute` а параметре метода Controller-а даёт доступ к конкретному атрибуту Модели

# Формы Spring MVC

**Dear Employee, Please enter your details**

Name

Surname

Salary

Department

OK

**Dear Employee, Please enter your details**

Name

Surname

Salary

Department

OK

# Формы Spring MVC

**form:select** – форма, предназначенная для реализации выпадающего списка

```
Department <form:select path="department">  
<form:option value="Information Technology" label="IT"/>  
<form:option value="Human Resources" label="HR"/>  
<form:option value="Sales" label="Sales"/>  
</form:select>
```

```
Department <form:select path="department">  
<form:options items="{employee.departments}"/>  
</form:select>
```

# Формы Spring MVC

**Dear Employee, Please enter your details**

Name

Surname

Salary

Department

Which car do you want? BMW ☐ Audi ☒ MB ☐

# Формы Spring MVC

**form:radiobutton** – форма, предназначенная для реализации radio button (переключатель)

Which car do you want?

BMW `<form:radiobutton path="carBrand" value="BMW"/>`

Audi `<form:radiobutton path="carBrand" value="Audi"/>`

MB `<form:radiobutton path="carBrand" value="Mercedes-Benz"/>`

Which car do you want?

`<form:radiobuttons path="carBrand" items="${employee.carBrands}"/>`

# Формы Spring MVC

**Dear Employee, Please enter your details**

Name

Surname

Salary

Department  ▼

Which car do you want? ☐ Audi ☐ MB ☐ BMW

Foreign Language(s) ☐ EN ☐ FR ☐ DE

# Формы Spring MVC

**form:checkbox** – форма, предназначенная для реализации check box (флажок)

```
Foreign Language(s)
EN <form:checkbox path="languages" value="English"/>
DE <form:checkbox path="languages" value="Deutch"/>
FR <form:checkbox path="languages" value="French"/>
```

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Languages:

```
<ul>
  <c:forEach var="lang" items="${employee.languages}">
    <li> ${lang} </li>
  </c:forEach>
</ul>
```

- ul – unordered list
- li – list item

```
Foreign Language(s)
<form:checkboxes path="languages" items="${employee.languagesList}"/>
```



# Валидация форм Spring MVC

**Dear Employee, Please enter your details**

Name

Surname

Salary

Department

Which car do you want? ☐ Audi ☐ MB ☐ BMW

Foreign Language(s) ☐ EN ☐ FR ☐ DE

# Hibernate

JPA (Java Persistence API) – это стандартная спецификация, которая описывает систему для управления сохранением Java объектов в таблицы базы данных

Hibernate – самая популярная реализация спецификации JPA

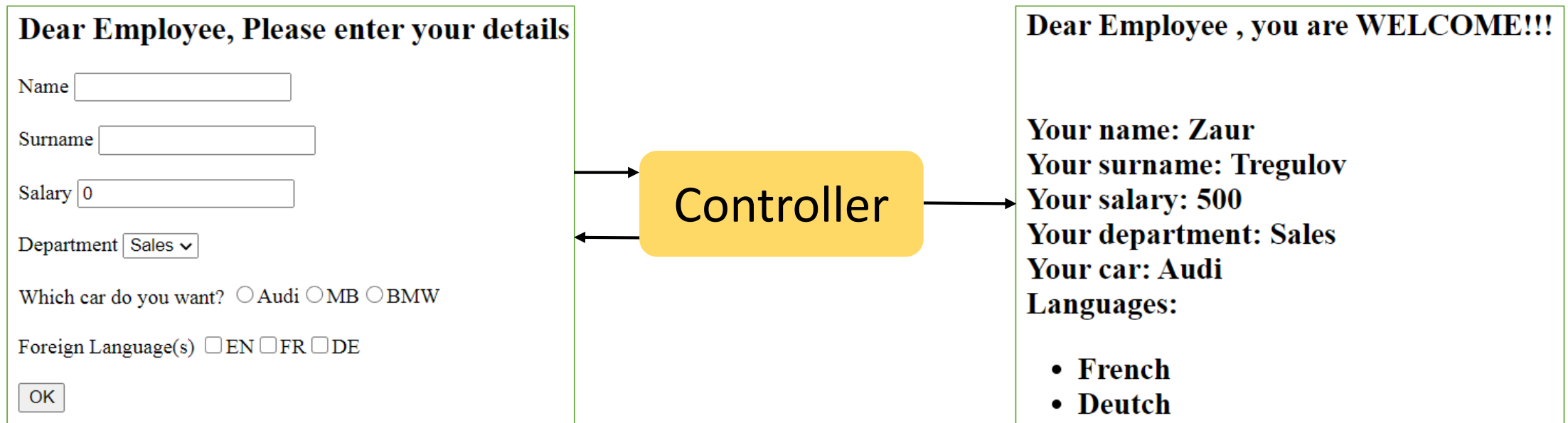
Таким образом JPA описывает правила, а Hibernate реализует их.

# Валидация форм Spring MVC

Java Standard Bean Validation API – это спецификация, которая описывает правила валидации

Hibernate Validator – реализация правил, описанных в Java Standard Bean Validation API

# Валидация форм Spring MVC



# Валидация форм Spring MVC

@Size – размер поля должен быть между заданными границами

```
@Size(min = 2, message = "name must be min 2 symbols")  
private String name;
```

@NotEmpty – поле не должно быть пустым

```
@NotEmpty(message = "surname is required field")  
private String surname;
```

@NotBlank – поле не должно быть пустым и не должно быть заполнено только пробелами

```
@NotBlank(message = "surname is required field")  
private String surname;
```

# Валидация форм Spring MVC

```
Name <form:input path="name"/>
<form:errors path="name"/>
```

**form:errors** - форма,  
отображающая ошибки

```
@RequestMapping("/showDetails")
public String showEmployeeDetails(
    @Valid @ModelAttribute("employee") Employee emp
    , BindingResult bindingResult) {

    if (bindingResult.hasErrors()) {
        return "ask-emp-details-view";
    } else {
        return "show-emp-details-view";
    }
}
```

**@Valid** означает, что  
атрибут должен пройти  
процесс валидации

**BindingResult** содержит  
результат этой валидации

# Валидация форм Spring MVC

@Min – числовое значение должно быть больше или равно указанного параметра

@Max – числовое значение должно быть меньше или равно указанному параметру

```
@Min(value = 500, message = "must be greater than 499")  
@Max(value = 1000, message = "must be less than 1001")  
private int salary;
```

# Валидация форм Spring MVC

@Pattern – значение поля должно соответствовать определённому Регулярному Выражению

```
@Pattern(regex = "\\d{3}-\\d{2}-\\d{2}"  
|           , message = "please use pattern XXX-XX-XX")  
private String phoneNumber;
```



# Spring MVC + Hibernate

## All Employees

Name	Surname	Department	Salary	Operations	
Zaur	Tregulov	IT	1500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Misha	Ivanov	Sales	800	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Oleg	Petrov	HR	2500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>

# Spring MVC + Hibernate

## Employee Info

Name

Surname

Department

Salary

OK

# Spring MVC + Hibernate

## Employee Info

Name

Surname

Department

Salary

# Spring MVC + Hibernate

Конфигурация приложения Spring MVC + Hibernate:

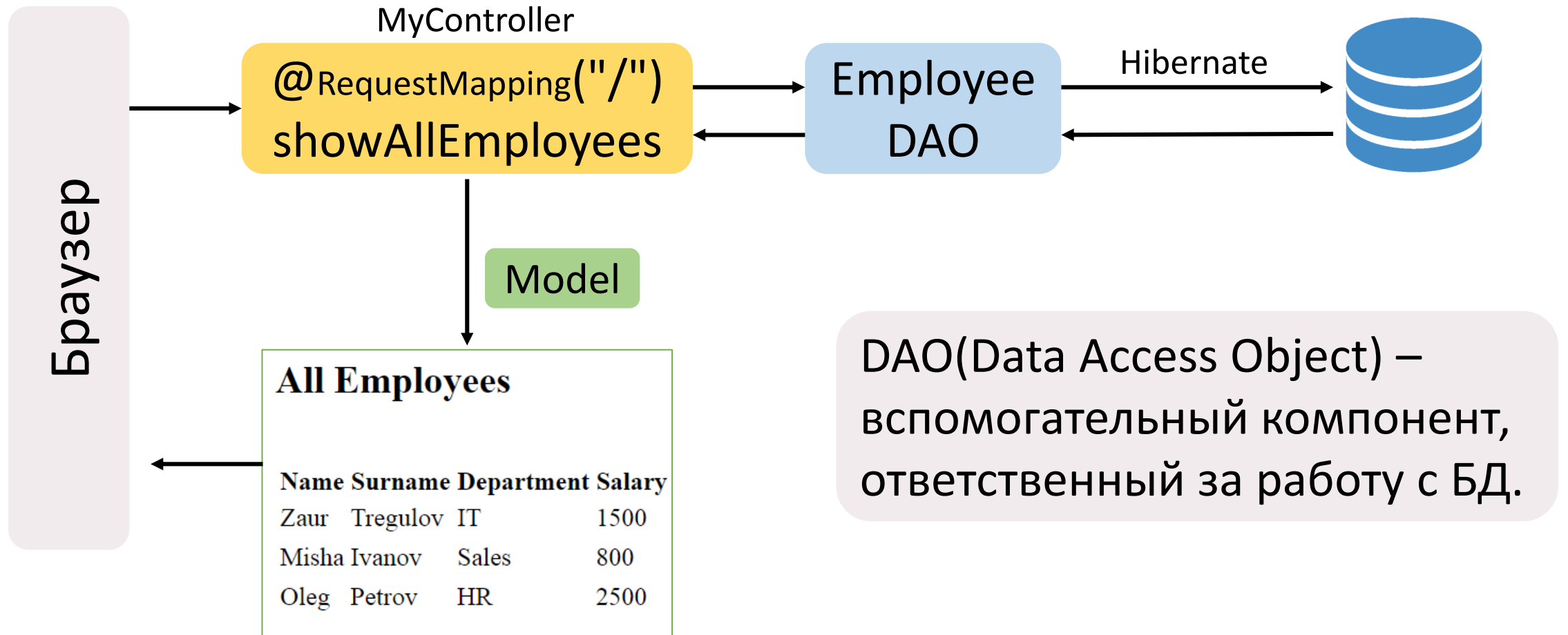
- Подготовка базы данных и таблицы
- Добавление dependency в pom файл
- Конфигурация web.xml
- Конфигурация applicationContext.xml

# Spring MVC + Hibernate

## All Employees

Name	Surname	Department	Salary
Zaur	Tregulov	IT	1500
Misha	Ivanov	Sales	800
Oleg	Petrov	HR	2500

# Spring MVC + Hibernate



# Spring MVC + Hibernate

## All Employees

Name	Surname	Department	Salary	Operations	
Zaur	Tregulov	IT	1500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Misha	Ivanov	Sales	800	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Oleg	Petrov	HR	2500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>

# Spring MVC + Hibernate

При использовании аннотации `@Transactional`, Spring берёт на себя ответственность за открытие и закрытие транзакций

`@Controller` – это специализированный `@Component`

`@Repository` – это специализированный `@Component`.

Данная аннотация используется для DAO.

При поиске компонентов, Spring также будет регистрировать все DAO с аннотацией `@Repository` в Spring Container-e

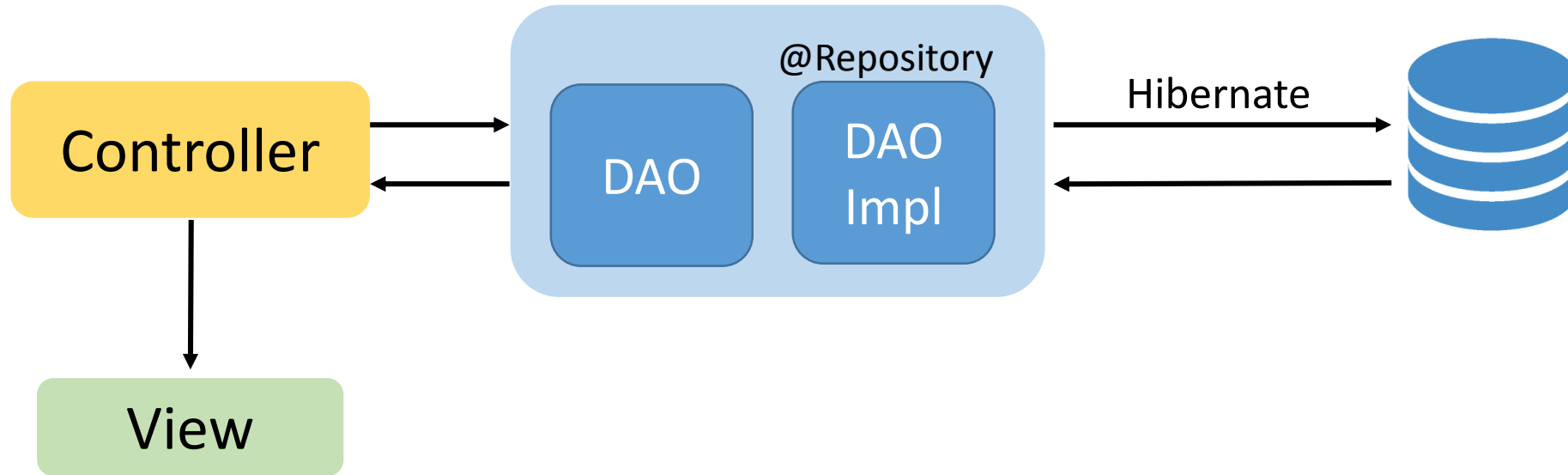


# Spring MVC + Hibernate

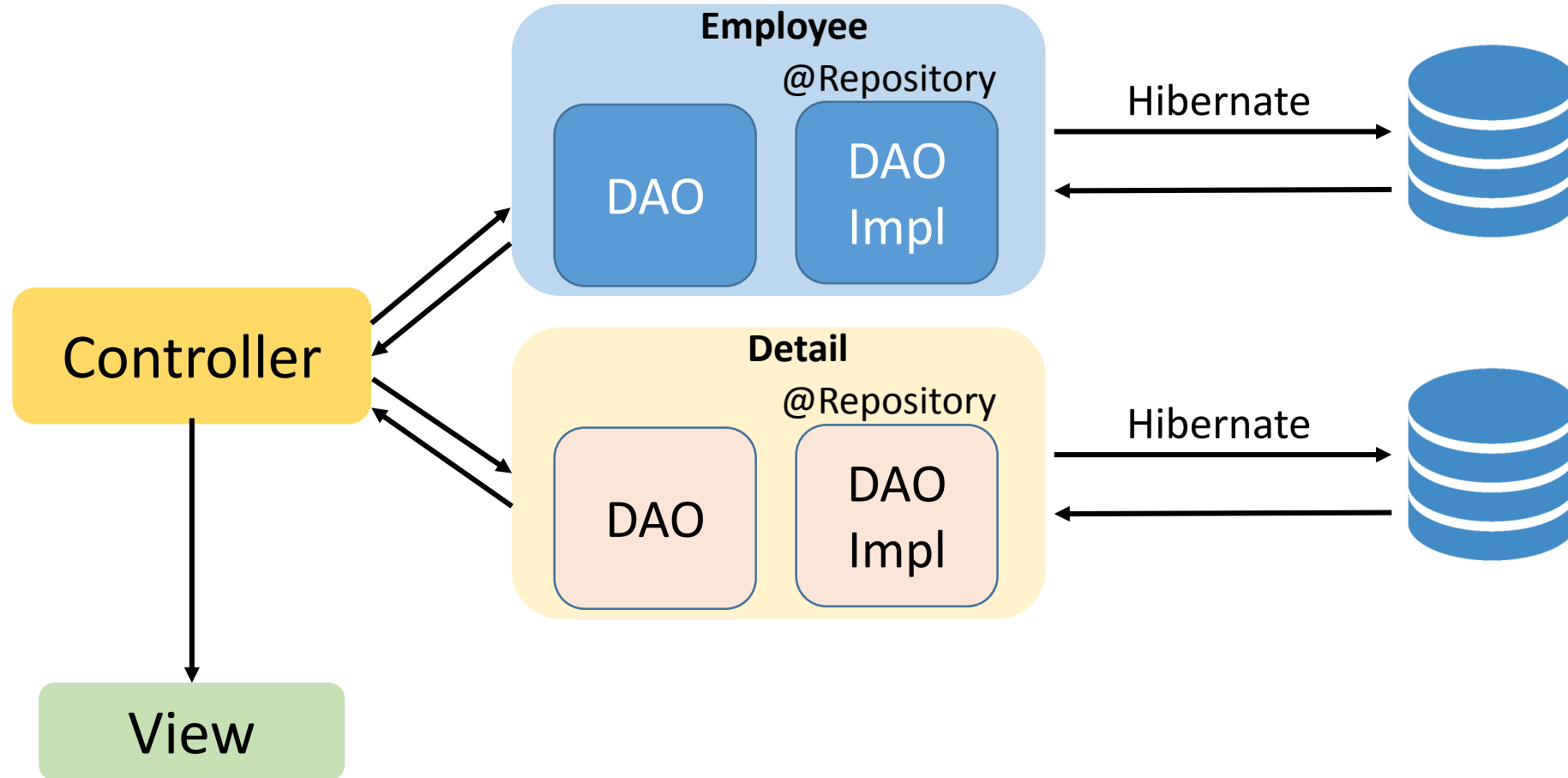
## All Employees

Name	Surname	Department	Salary
Zaur	Tregulov	IT	1500
Misha	Ivanov	Sales	800
Oleg	Petrov	HR	2500

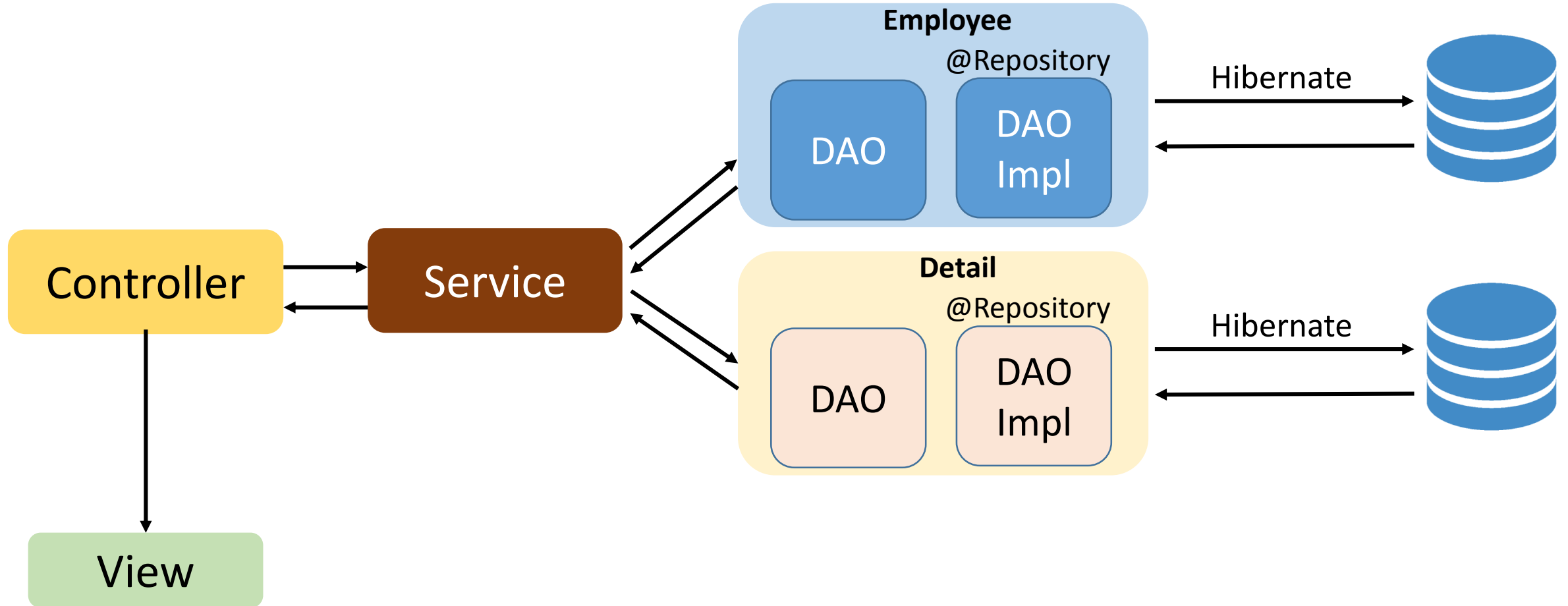
# Spring MVC + Hibernate



# Spring MVC + Hibernate



# Spring MVC + Hibernate



# Spring MVC + Hibernate

Аннотация `@Service` отмечает класс, содержащий бизнес-логику.

В иерархии компонентов приложения `Service` является соединительным звеном между `Controller`-ом и `DAO`

`@Service` – это специализированный `@Component`.

При поиске компонентов, Spring также будет регистрировать все классы с аннотацией `@Service` в Spring Container-е

# Spring MVC + Hibernate

## All Employees

Name	Surname	Department	Salary	Operations	
Zaur	Tregulov	IT	1500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Misha	Ivanov	Sales	800	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Oleg	Petrov	HR	2500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>

# Spring MVC + Hibernate

## Employee Info

Name

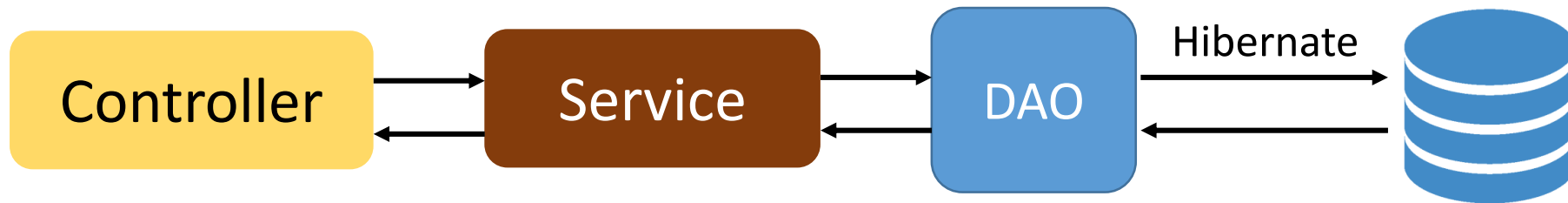
Surname

Department

Salary

OK

# Spring MVC + Hibernate





# Spring MVC + Hibernate

## All Employees

Name	Surname	Department	Salary	Operations	
Zaur	Tregulov	IT	1500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Misha	Ivanov	Sales	800	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Oleg	Petrov	HR	2500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>

# Spring MVC + Hibernate

## Employee Info

Name

Surname

Department

Salary

# Spring MVC + Hibernate

## All Employees

Name	Surname	Department	Salary	Operations	
Zaur	Tregulov	IT	1500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Misha	Ivanov	Sales	800	<input type="button" value="Update"/>	<input type="button" value="Delete"/>
Oleg	Petrov	HR	2500	<input type="button" value="Update"/>	<input type="button" value="Delete"/>

# Spring MVC + Hibernate + AOP

## Конфигурация AOP

- Добавление dependency в pom файл
- Добавление конфигурации в applicationContext.xml

# REST API

REST – REpresentational State Transfer



# REST API



REST – это очень удобный способ коммуникации между приложениями. Он описывает стандарты, используя которые Клиент взаимодействует с Сервером посредством HTTP протокола

# REST API

Вызовы REST API могут осуществляться, используя HTTP протокол

REST API Не принуждает использовать какой-то конкретный язык программирования.  
Клиентская и серверная части нашего приложения могут быть написаны на разных языках программирования

Для передачи информации можно использовать не только JSON, но и любой другой формат данных

# JSON

JSON – JavaScript Data Notation

Формат данных JSON представляет собой текстовую информацию

JSON используется для хранения и особенно для обмена информацией

JSON не привязан к какому-то конкретному языку программирования и используется повсеместно

Формат данных JSON содержит коллекцию пар ключ-значение

JSON Data Binding или JSON Mapping – это привязка JSON к Java объекту



# JSON

```
class Car {  
    private String model;  
    private String color;  
  
    public Car() {  
    }  
  
    //getters and setters
```

```
class House {  
    private int numberOfRooms;  
  
    public House() {  
    }  
  
    //getters and setters
```

```
class Employee {  
    private String name;  
    private String surname;  
    private int salary;  
    private String[] languages;  
    private Car car;  
    private House house;  
  
    public Employee() {  
    }  
  
    //getters and setters
```

# JSON

getters

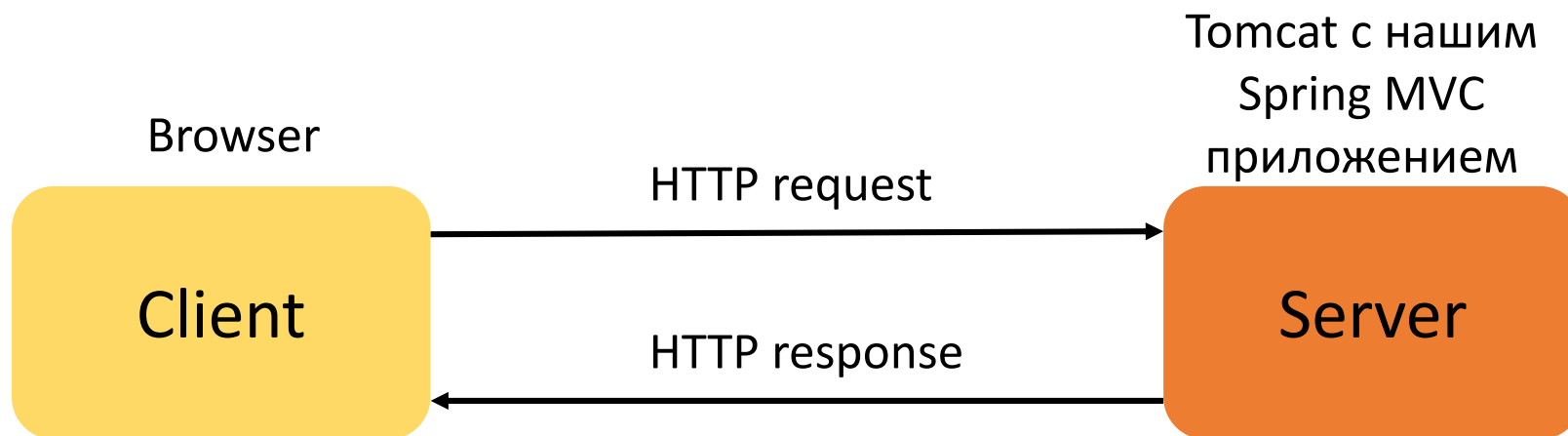
```
Employee emp = new Employee();  
emp.setName("Zaur");  
emp.setSurname("Tregulov");  
emp.setSalary(750);  
emp.setLanguages(new String[]{"English",  
    , "Deutch", "French"});  
Car car = new Car();  
car.setModel("BMW");  
car.setColor("red");  
emp.setCar(car);
```

```
{  
  "name": "Zaur",  
  "surname": "Tregulov",  
  "salary": 750,  
  "languages": ["English", "Deutch", "French"],  
  "car": {  
    "model": "BMW",  
    "color": "red"  
  },  
  "house": null  
}
```

setters

# HTTP протокол

HTTP – это протокол, который используется для передачи данных в сети.



# HTTP протокол

## All Employees

Name	Surname	Department	Salary	Operations	
Zaur	Tregulov	IT	1500	<button>Update</button>	<button>Delete</button>
Misha	Ivanov	Sales	800	<button>Update</button>	<button>Delete</button>
Oleg	Petrov	HR	2500	<button>Update</button>	<button>Delete</button>

Add

# HTTP протокол

## Employee Info

Name

Surname

Department

Salary

# HTTP протокол

@GetMapping связывает HTTP запрос, использующий HTTP метод GET с методом контроллера.

@PostMapping связывает HTTP запрос, использующий HTTP метод POST с методом контроллера.

## GET

- Передаваемая информация хранится в URL
- Ограничен максимальной длиной
- Не поддерживает передачу бинарных данных
- Можно поделиться ссылкой
- Как правило, используется для получения информации

## POST

- Передаваемая информация хранится в теле запроса
- Не ограничен максимальной длиной
- Поддерживает передачу бинарных данных
- Ссылкой поделиться нельзя
- Как правило, используется для добавления данных

# HTTP request

Request line

HTTP метод и адрес (URL)

Zero or more Headers

Метаданные о запросе

An empty line

Разделяет отдел header от  
отдела body

Message body(optional)

Payload – полезная нагрузка

# HTTP response

Status line

Код статуса и текст статуса

Zero or more Headers

Метаданные об ответе

An empty line

Разделяет отдел header от  
отдела body

Message body(optional)

Payload – полезная нагрузка



# HTTP response status codes

**1XX** – Informational. Запрос был получен и процесс продолжается.

**2XX** – Success. Запрос был успешно получен, понят и принят.

**3XX** – Redirection. Для выполнения запроса необходимо предпринять дальнейшие действия.

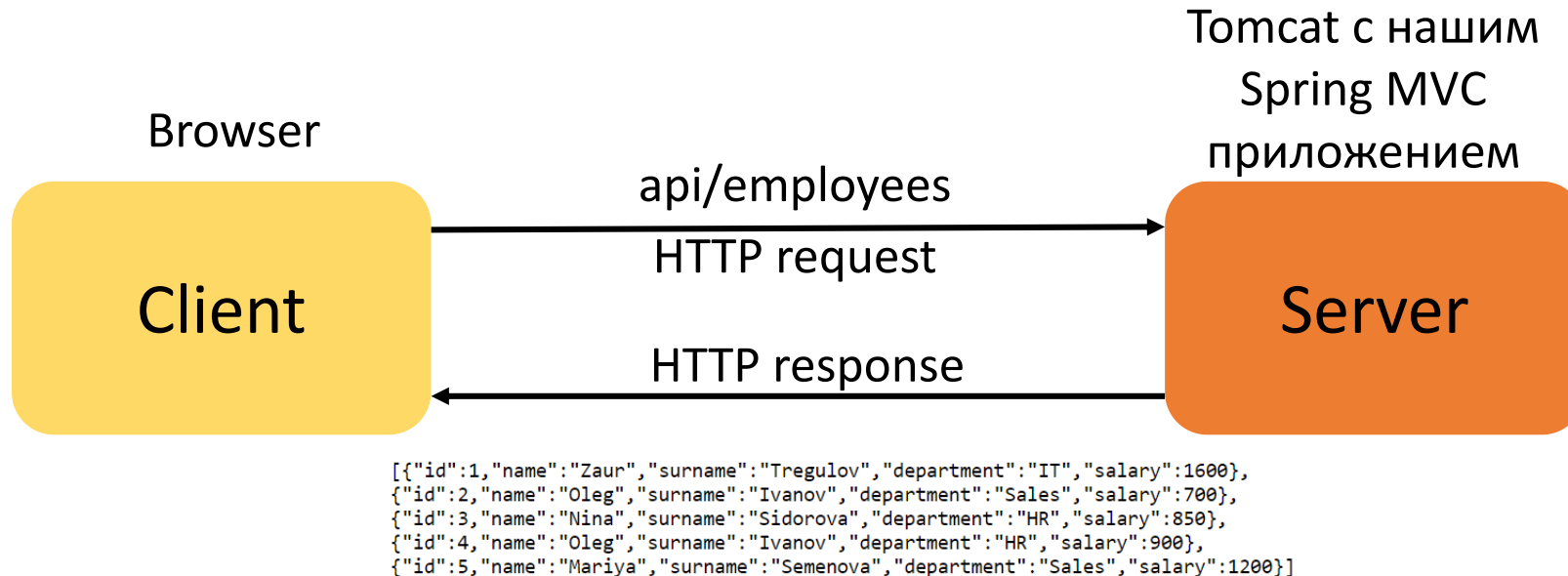
**4XX** – Client Error. Запрос содержит неверный синтаксис или не может быть выполнен.

404 File Not Found

**5XX** – Server Error. Серверу не удалось выполнить корректный запрос.

500 Internal  
Server Error

# REST API



# REST API

При создании REST API настоятельно рекомендуется придерживаться общепринятых стандартов:

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

# REST API

Конфигурация приложения Spring MVC + Hibernate без участия XML файлов:

- Подготовка базы данных и таблицы
- Добавление dependency в pom файл
- Создание конфигурационного Java класса
- Создание класса для реализации Dispatcher Servlet
- Добавление в проект Tomcat

# REST API

@RestController – это Controller, который управляет REST запросами и ответами.

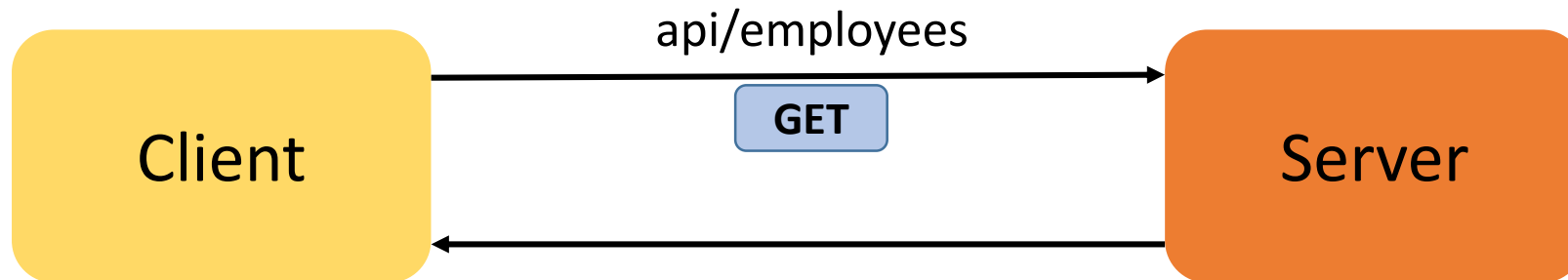
```
@RestController  
@RequestMapping("/api")  
public class MyRestController {
```

# REST API

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

# REST API

Получение всех работников



```
[{"id":1,"name":"Zaur","surname":"Tregulov","department":"IT","salary":1600},  
{"id":2,"name":"Oleg","surname":"Ivanov","department":"Sales","salary":700},  
{"id":3,"name":"Nina","surname":"Sidorova","department":"HR","salary":850},  
{"id":4,"name":"Oleg","surname":"Ivanov","department":"HR","salary":900},  
{"id":5,"name":"Mariya","surname":"Semenova","department":"Sales","salary":1200}]
```

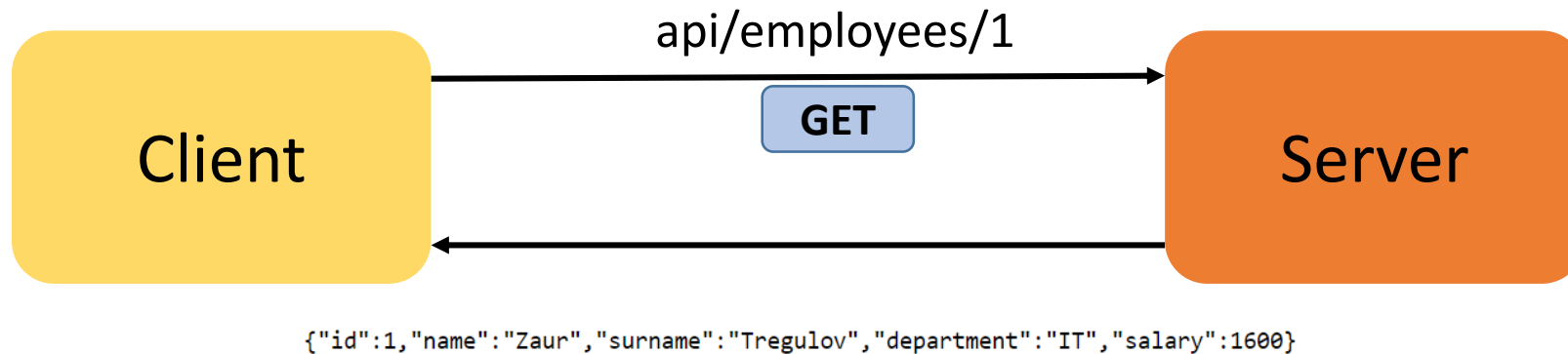
# REST API

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника



# REST API

Получение одного работника

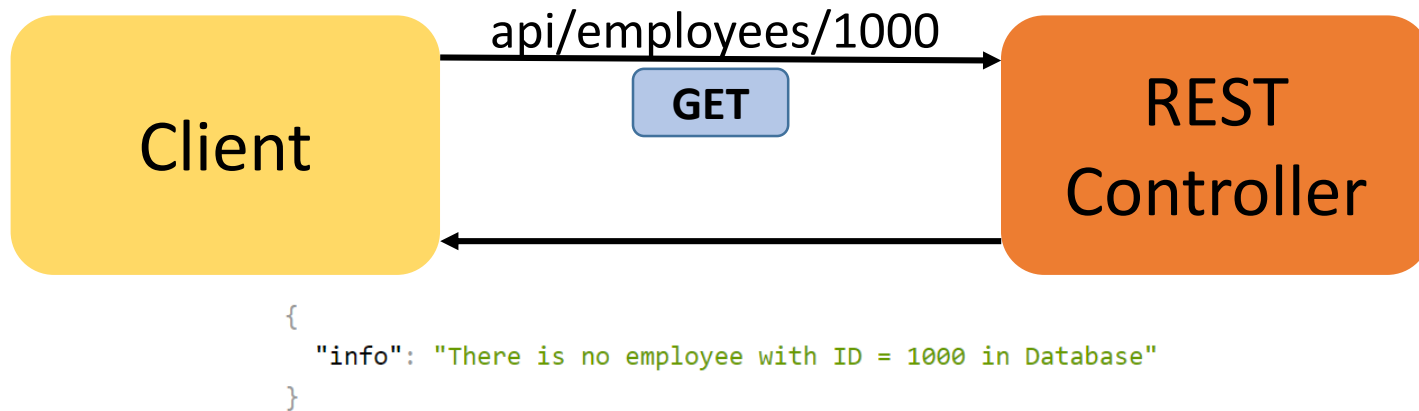


# REST API

Аннотация `@PathVariable` используется для получения значения переменной из адреса запроса

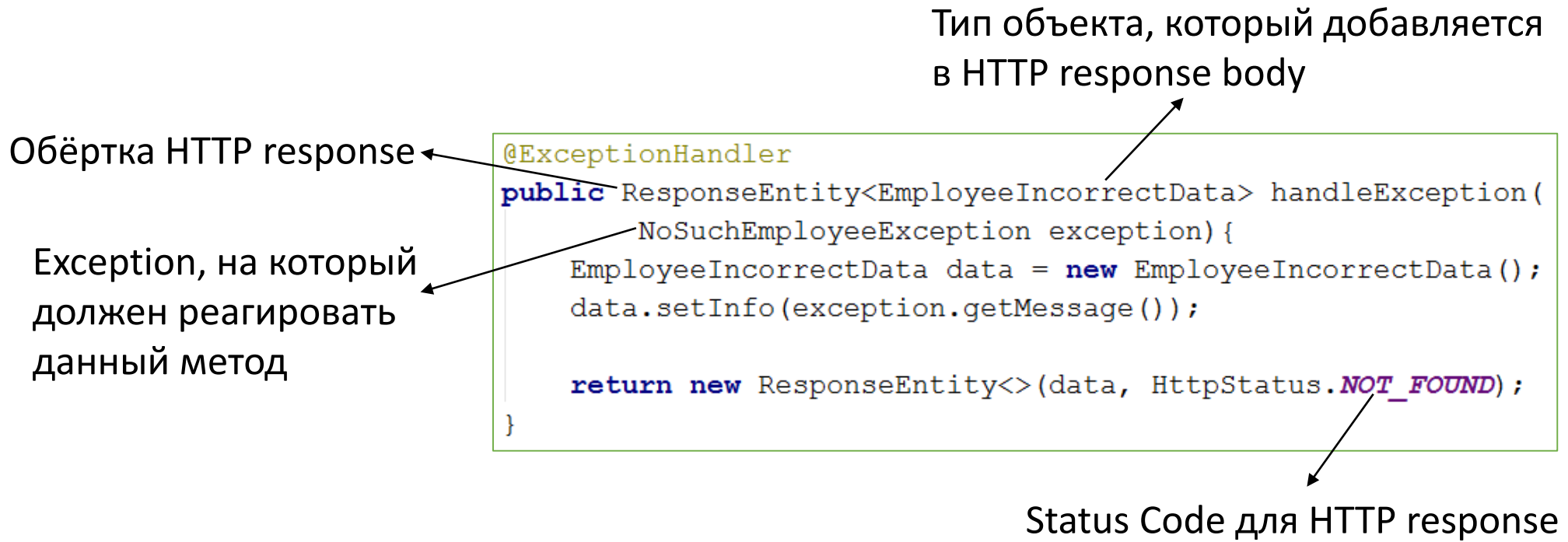
```
@GetMapping("/employees/{id}")  
public Employee getEmployee(@PathVariable int id) {  
    Employee employee = employeeService.getEmployee(id);  
    return employee;  
}
```

# REST API

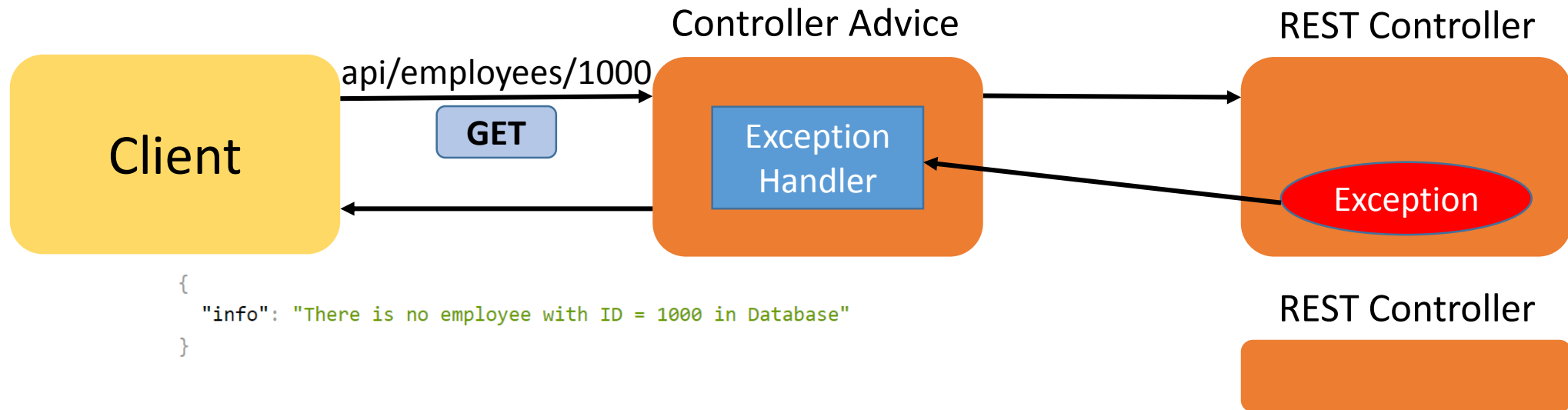


# REST API

Аннотацией `@ExceptionHandler` отмечается метод, ответственный за обработку исключений



# REST API



Аннотацией `@ControllerAdvice` отмечается класс, предоставляющий функциональность Global Exception Handler-а

# REST API

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

# Spring MVC + Hibernate

## Employee Info

Name

Surname

Department

Salary

OK

# REST API

## Добавление работника





# REST API

Аннотация `@PostMapping` связывает HTTP запрос, использующий HTTP метод POST с методом контроллера.

Аннотация `@RequestBody` связывает тело HTTP метода с параметром метода Controller-a

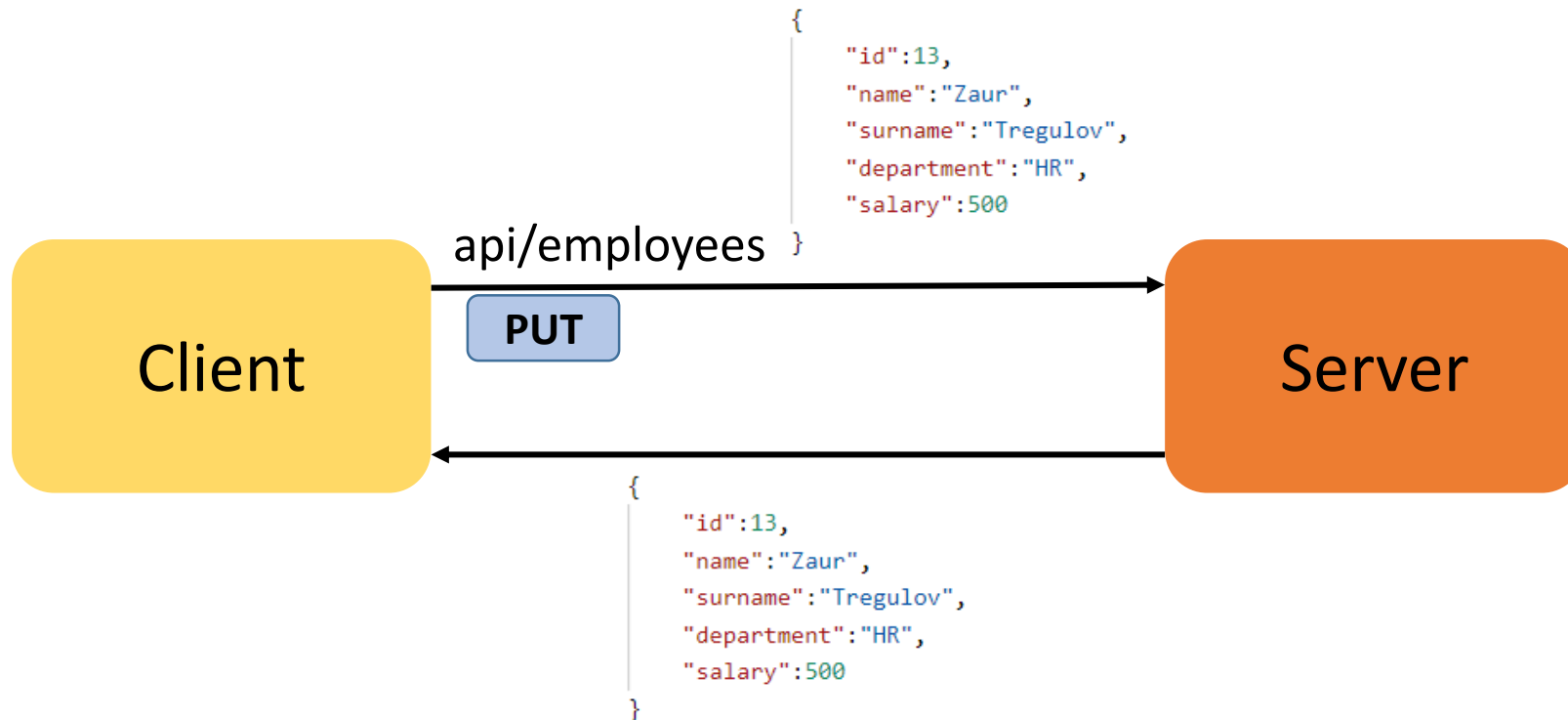
```
@PostMapping("/employees")
public Employee addNewEmployee(@RequestBody Employee employee) {
    employeeService.saveEmployee(employee);
    return employee;
}
```

# REST API

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

# REST API

## Изменение работника



# REST API

Аннотация `@PutMapping` связывает HTTP запрос, использующий HTTP метод PUT с методом контроллера.

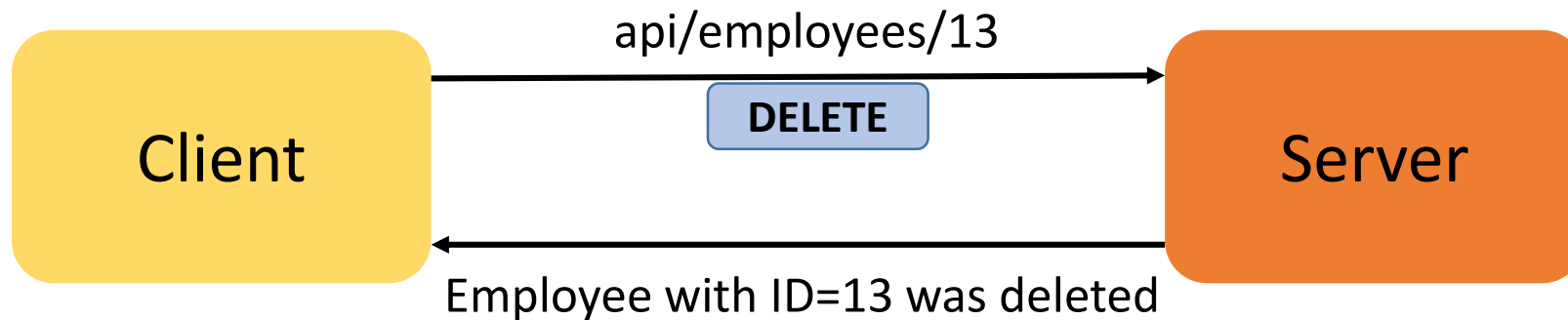
```
@PutMapping("/employees")
public Employee updateEmployee(@RequestBody Employee employee) {
    employeeService.saveEmployee(employee);
    return employee;
}
```

# REST API

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

# REST API

Удаление работника



# REST API

Аннотация `@DeleteMapping` связывает HTTP запрос, использующий HTTP метод DELETE с методом контроллера.

```
@DeleteMapping("/employees/{id}")  
public String deleteEmployee(@PathVariable int id){  
    employeeService.deleteEmployee(id);  
    return "Employee with ID = " + id + " was deleted.";  
}
```

# REST API

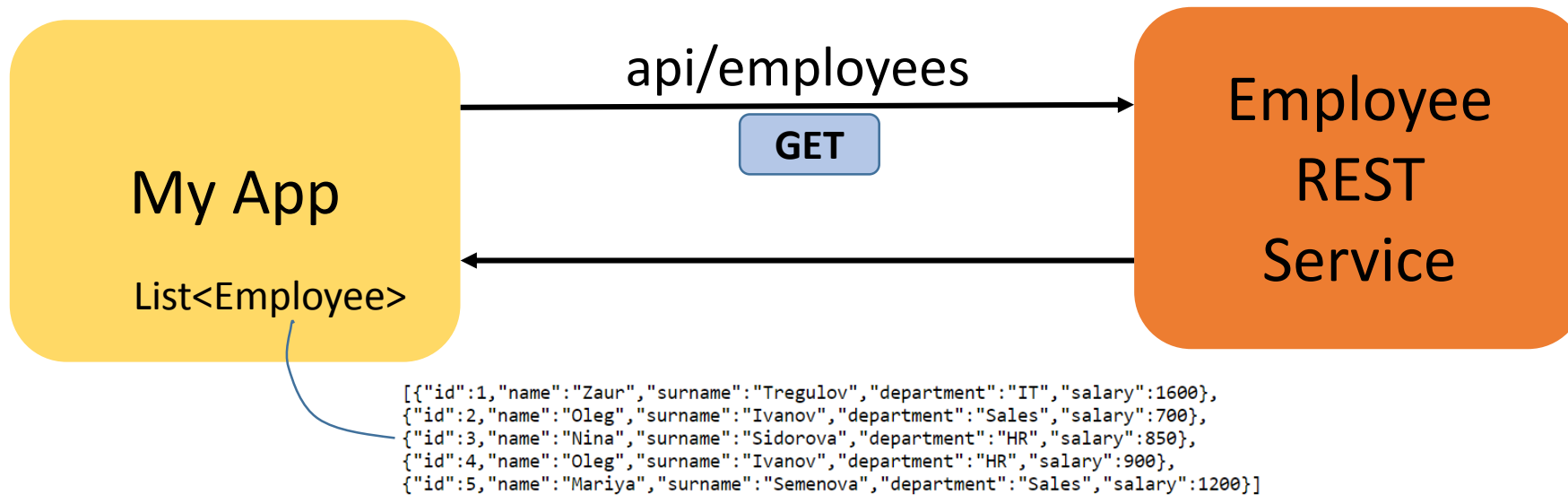




# REST API

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

# REST API



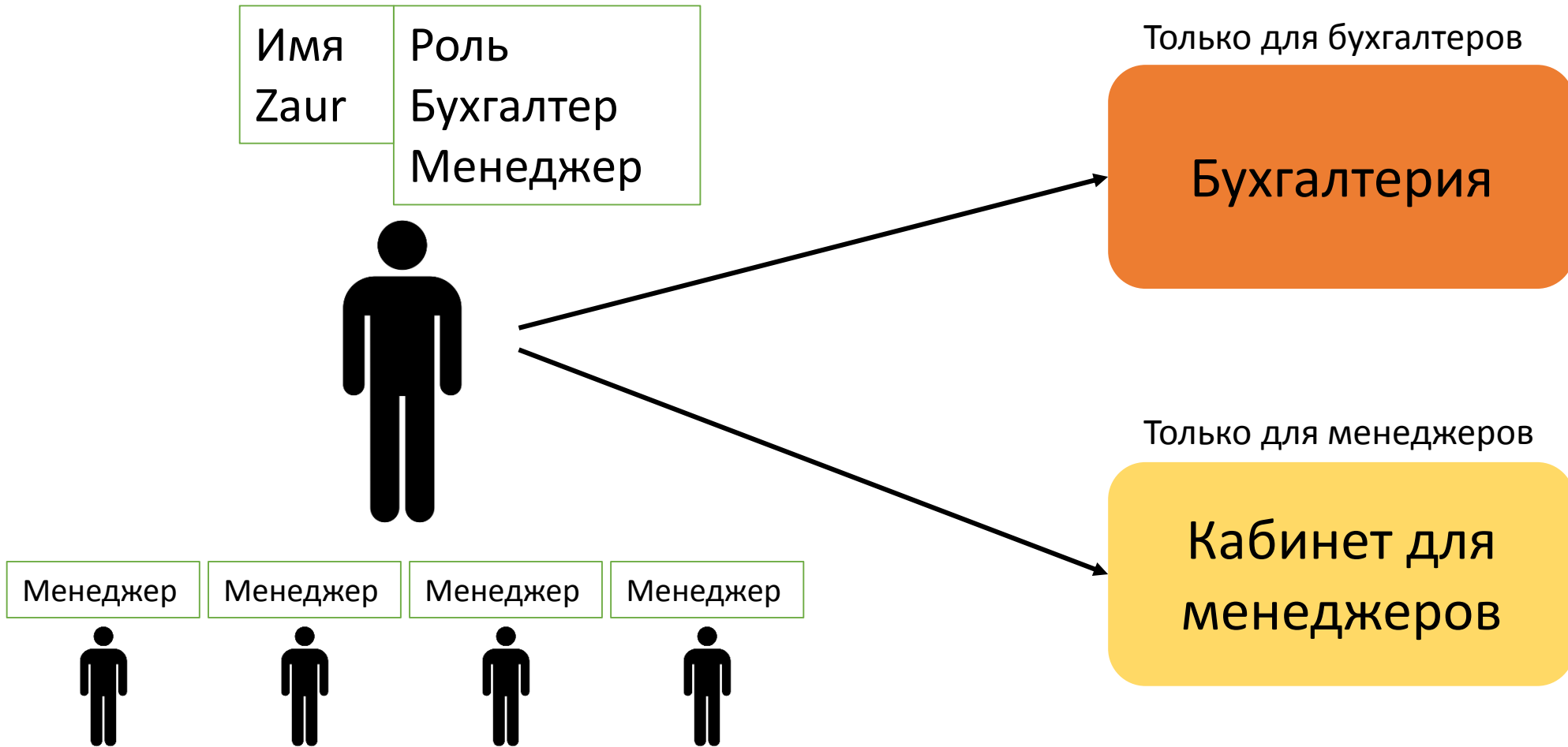
# Spring Security

Spring Security предоставляет функционал для обеспечения безопасности приложения

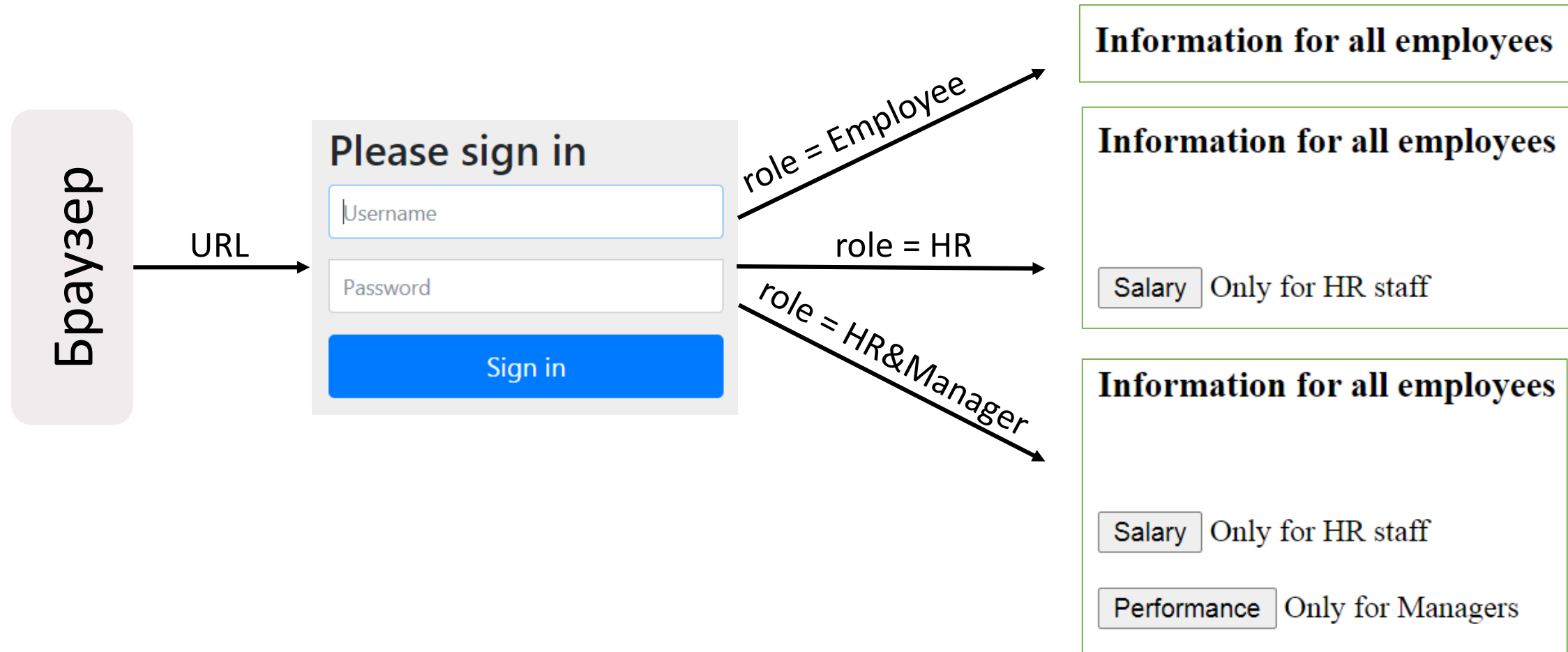
Аутентификация – процедура проверки подлинности путём сравнения введённых user name-а и пароля с user name-ом и паролем, хранящимися в базе

Авторизация – процедура проверки разрешений на доступ к тому или иному ресурсу

# Spring Security



# Spring Security

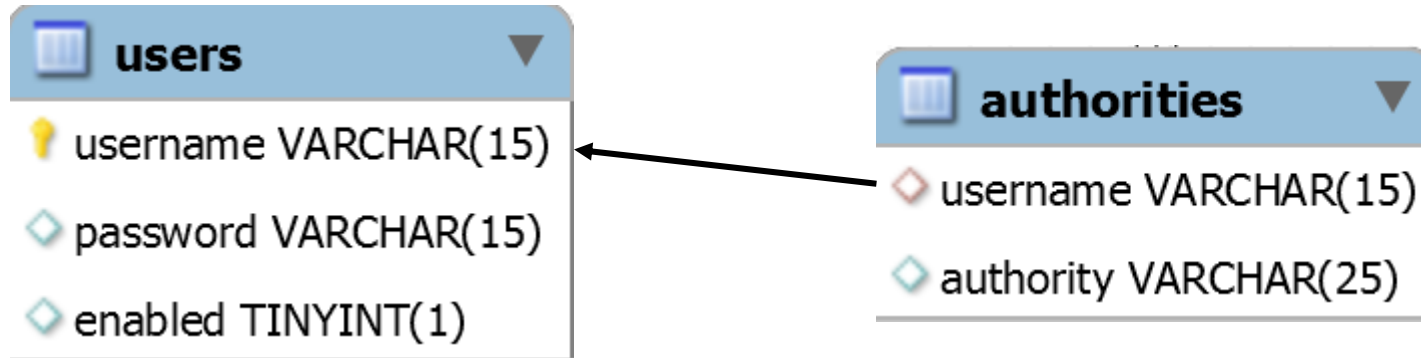


# Spring Security

Конфигурация приложения Spring MVC + Spring Security без участия XML файлов:

- Добавление dependency в pom файл
- Создание конфигурационного Java класса
- Создание класса для реализации Dispatcher Servlet
- Добавление в проект Tomcat
- Создание Security Initializer
- Создание конфигурации для Spring Security

# Spring Security



Если `enabled = 1`, это означает, что юзер может совершать вход в систему

Если `enabled = 0`, это означает, что юзер заблокирован

# Spring Security

Пароль в таблице содержится в следующем виде:

**{алгоритм кодирования}** зашифрованный пароль

- **{noop}** – никакого шифрования. Пароль прописывается в виде обычного текста.
- **{bcrypt}** – шифрование с помощью хеш-функции bcrypt. Пароль прописывается в зашифрованном виде.



# Spring Security

**bcrypt**(текстовый пароль + соль) = шифрованный пароль

# Spring Boot

Spring Boot – это фреймворк, целью которого является упрощение создания приложения на основе Spring

Spring Boot облегчает работу программиста по следующим направлениям:

- Автоматическая конфигурация приложения
- Управление зависимостями в maven проектах
- Встроенная поддержка сервера

# Spring Boot

Starter пакет – это набор взаимосвязанных зависимостей, которые могут быть добавлены в приложение.

@SpringBootApplication

=

@Configuration

+

@EnableAutoConfiguration

+

@ComponentScan

# Spring Boot

## Spring Boot REST API

HTTP метод	URL	CRUD операция
GET	api/employees	Получение всех работников
GET	api/employees/{employeeId}	Получение одного работника
POST	api/employees	Добавление работника
PUT	api/employees	Изменение работника
DELETE	api/employees/{employeeId}	Удаление работника

# Spring Boot

## Конфигурация приложения

- Подготовка базы данных и таблицы
- Добавление dependency в pom файл
- Добавление properties для подключения к базе данных

# Spring Boot

JPA (Java Persistence API) – это стандартная спецификация, которая описывает систему для управления сохранением Java объектов в таблицы базы данных

Hibernate – самая популярная реализация спецификации JPA

Таким образом JPA описывает правила, а Hibernate реализует их.

# Spring Boot

В Spring Boot дефолтной реализацией JPA является Hibernate.

Роль Hibernate сессии в JPA выполняет EntityManager

# Spring Boot

Spring Data JPA – удобный механизм для взаимодействия с таблицами баз данных, позволяющий минимизировать количество кода

Метод	CRUD операция
findAll	Получение всех работников
findById	Получение одного работника
save	Добавление работника
save	Изменение работника
deleteById	Удаление работника



# Spring Boot

Spring Data REST предоставляет механизм автоматического создания REST API на основе типа Entity, прописанного в репозитории проекта

HTTP метод	URL	CRUD операция
PUT	employees/ <b>id</b>	Изменение работника

# Spring Boot

Spring Boot Actuator предоставляет готовые конечные точки (endpoints), с помощью которых мониторится приложение

Адрес	Описание
/actuator/ <b>health</b>	Информация о статусе приложения
/actuator/ <b>info</b>	Информация о приложении
/actuator/ <b>beans</b>	Информация о всех бинах, зарегистрированных в Spring Container-е
/actuator/ <b>mappings</b>	Информация о всех Mapping-ах

**Spring**  
**Hibernate**  
**Spring Boot**  
**для**  
**начинающих**



**zaurtregulov@gmail.com**