Create JPA DAO in Spring Boot

Various DAO Techniques

Various DAO Techniques

- Version 1: Use EntityManager but leverage native Hibernate API

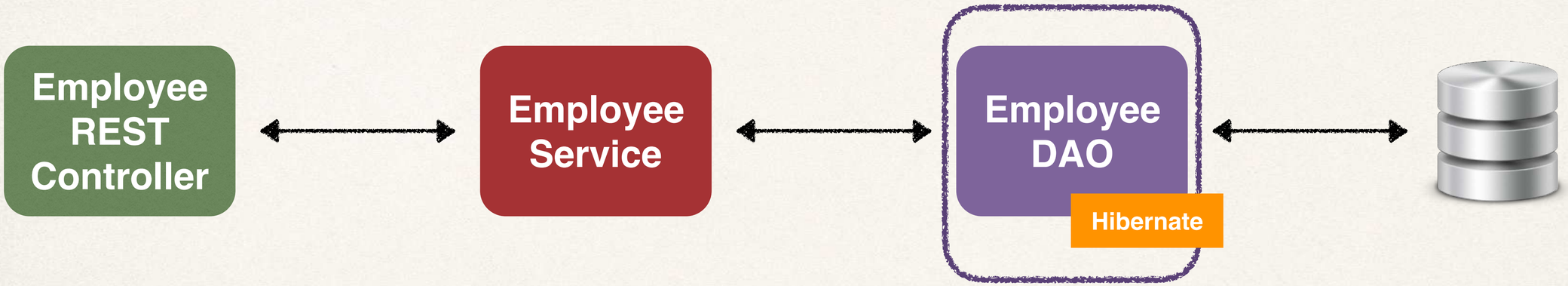
Various DAO Techniques

- Version 1: Use EntityManager but leverage native Hibernate API
- Version 2: Use EntityManager and standard JPA API

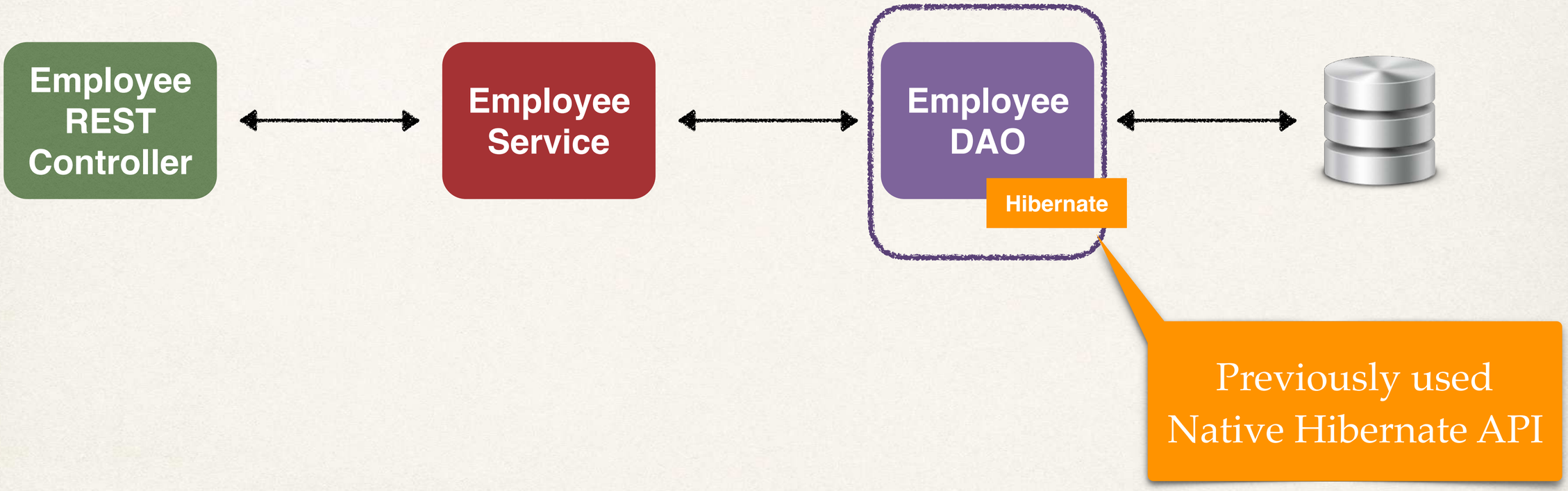
Various DAO Techniques

- Version 1: Use EntityManager but leverage native Hibernate API
- Version 2: Use EntityManager and standard JPA API
- Version 3: Spring Data JPA

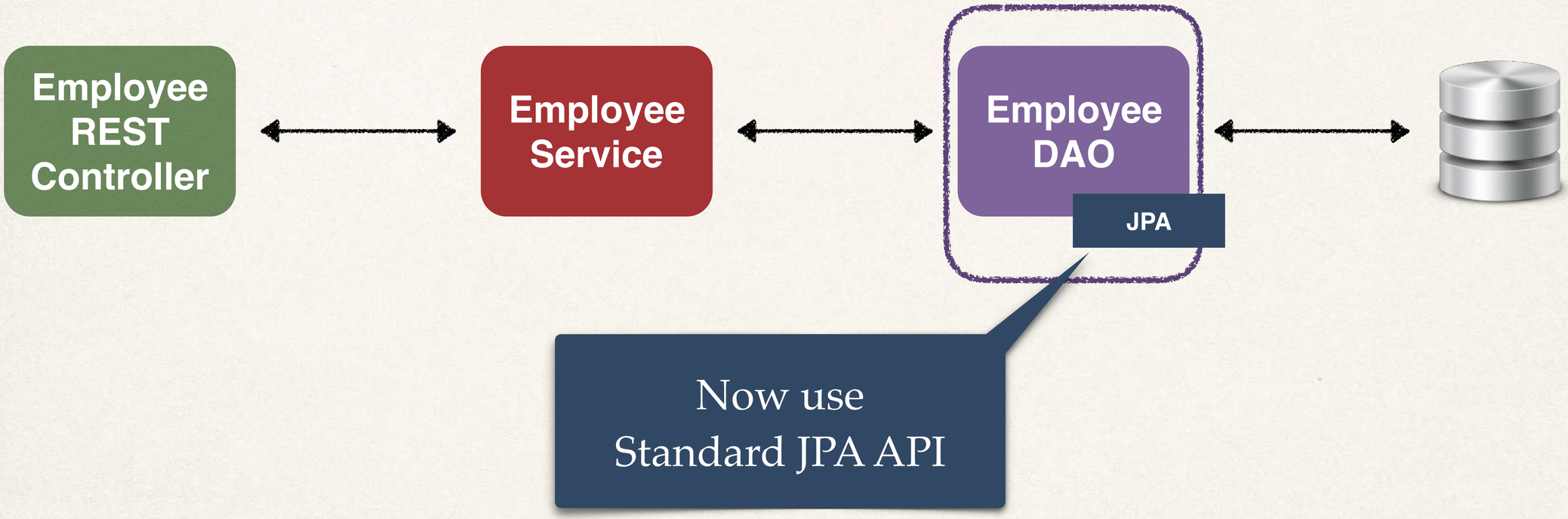
Application Architecture



Application Architecture



Application Architecture



Remember the Benefits of JPA

Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation

Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code

Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code
- Can theoretically switch vendor implementations

Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code
- Can theoretically switch vendor implementations
  - If Vendor ABC stops supporting their product

Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code
- Can theoretically switch vendor implementations
  - If Vendor ABC stops supporting their product
  - Switch to Vendor XYZ without vendor lock in

Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation

My Biz App

- Maintain portable, flexible code

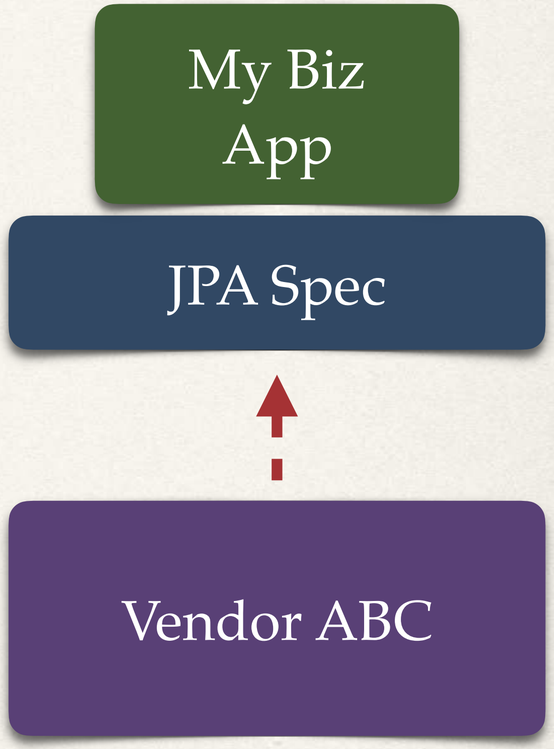
- Can theoretically switch vendor implementations
  - If Vendor ABC stops supporting their product
  - Switch to Vendor XYZ without vendor lock in

Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code
- Can theoretically switch vendor implementations
  - If Vendor ABC stops supporting their product
  - Switch to Vendor XYZ without vendor lock in

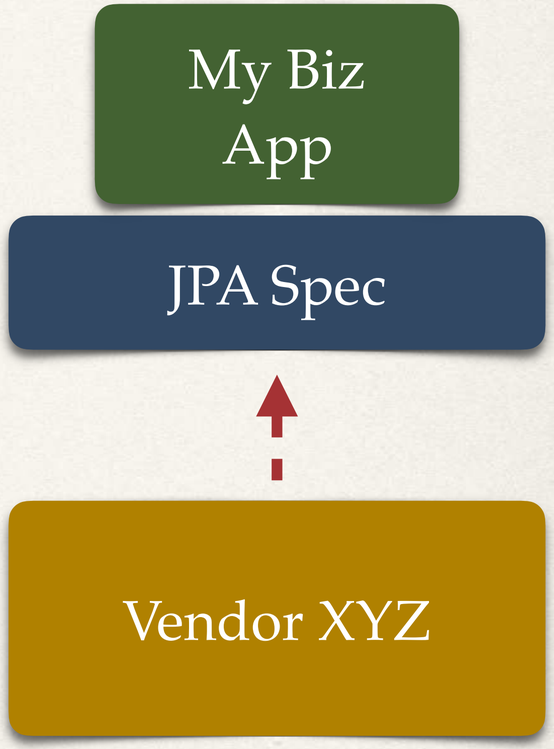
Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code
- Can theoretically switch vendor implementations
  - If Vendor ABC stops supporting their product
  - Switch to Vendor XYZ without vendor lock in



Remember the Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code
- Can theoretically switch vendor implementations
  - If Vendor ABC stops supporting their product
  - Switch to Vendor XYZ without vendor lock in



Standard JPA API

Standard JPA API

- The JPA API methods are similar to Native Hibernate API

Standard JPA API

- The JPA API methods are similar to Native Hibernate API

- JPA also supports a query language: JPQL (JPA Query Language)

Standard JPA API

- The JPA API methods are similar to Native Hibernate API

- JPA also supports a query language: JPQL (JPA Query Language)
  - For more details on JPQL, see this link

Standard JPA API

- The JPA API methods are similar to Native Hibernate API

- JPA also supports a query language: JPQL (JPA Query Language)
  - For more details on JPQL, see this link

www.luv2code.com/jpql

Comparing JPA to Native Hibernate Methods

Comparing JPA to Native Hibernate Methods

Action	Native Hibernate method	JPA method
Create/save new entity	`session.save(…)`	`entityManager.persist(…)`

Comparing JPA to Native Hibernate Methods

Action	Native Hibernate method	JPA method
Create/save new entity	`session.save(…)`	`entityManager.persist(…)`
Retrieve entity by id	`session.get(…) / load(…)`	`entityManager.find(…)`

Comparing JPA to Native Hibernate Methods

Action	Native Hibernate method	JPA method
Create/save new entity	`session.save(…)`	`entityManager.persist(…)`
Retrieve entity by id	`session.get(…) / load(…)`	`entityManager.find(…)`
Retrieve list of entities	`session.createQuery(…)`	`entityManager.createQuery(…)`

Comparing JPA to Native Hibernate Methods

Action	Native Hibernate method	JPA method
Create/save new entity	`session.save(…)`	`entityManager.persist(…)`
Retrieve entity by id	`session.get(…) / load(…)`	`entityManager.find(…)`
Retrieve list of entities	`session.createQuery(…)`	`entityManager.createQuery(…)`
Save or update entity	`session.saveOrUpdate(…)`	`entityManager.merge(…)`

Comparing JPA to Native Hibernate Methods

Action	Native Hibernate method	JPA method
Create/save new entity	`session.save(…)`	`entityManager.persist(…)`
Retrieve entity by id	`session.get(…) / load(…)`	`entityManager.find(…)`
Retrieve list of entities	`session.createQuery(…)`	`entityManager.createQuery(…)`
Save or update entity	`session.saveOrUpdate(…)`	`entityManager.merge(…)`
Delete entity	`session.delete(…)`	`entityManager.remove(…)`

Comparing JPA to Native Hibernate Methods

Action	Native Hibernate method	JPA method
Create/save new entity	`session.save(…)`	`entityManager.persist(…)`
Retrieve entity by id	`session.get(…) / load(…)`	`entityManager.find(…)`
Retrieve list of entities	`session.createQuery(…)`	`entityManager.createQuery(…)`
Save or update entity	`session.saveOrUpdate(…)`	`entityManager.merge(…)`
Delete entity	`session.delete(…)`	`entityManager.remove(…)`

High-level comparison
Other options depending on context …

Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Step-By-Step

Development Process

Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Step-By-Step

Let's build a DAO layer for this

Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Step-By-Step

Let's build a DAO layer for this

Using Standard JPA API

DAO Impl

DAO Impl

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {
```

DAO Impl

Same interface for consistent API

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {
```

DAO Impl

Same interface for consistent API

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {

  private EntityManager entityManager;
```

DAO Impl

Same interface for consistent API

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {

  private EntityManager entityManager;

  @Autowired
  public EmployeeDAOJpaImpl(EntityManager theEntityManager) {
    entityManager = theEntityManager;
  }

 ...
}
```

DAO Impl

Same interface for consistent API

```
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {

  private EntityManager entityManager;

  @Autowired
  public EmployeeDAOJpaImpl(EntityManager theEntityManager) {
    entityManager = theEntityManager;
  }

 ...
}
```

Automatically created by Spring Boot

Constructor injection

Get a list of employees

JPA API

Get a list of employees

JPA API

```
@Override
public List<Employee> findAll() {
```

Get a list of employees

JPA API

```
@Override
public List<Employee> findAll() {

  // create a query
  TypedQuery<Employee> theQuery =
      entityManager.createQuery("from Employee", Employee.class);
```

Get a list of employees

JPA API

```
@Override
public List<Employee> findAll() {

  // create a query
  TypedQuery<Employee> theQuery =
      entityManager.createQuery("from Employee", Employee.class);

  // execute query and get result list
  List<Employee> employees = theQuery.getResultList();
```

Get a list of employees

JPA API

```
@Override
public List<Employee> findAll() {

  // create a query
  TypedQuery<Employee> theQuery =
      entityManager.createQuery("from Employee", Employee.class);

  // execute query and get result list
  List<Employee> employees = theQuery.getResultList();

  // return the results
  return employees;
}
```

Using Standard JPA API

Get a list of employees

JPA API

Remember: No need to manage transactions …

Handled at Service layer with @Transactional

```
@Override
public List<Employee> findAll() {

  // create a query
  TypedQuery<Employee> theQuery =
      entityManager.createQuery("from Employee", Employee.class);

  // execute query and get result list
  List<Employee> employees = theQuery.getResultList();

  // return the results
  return employees;
}
```

Using Standard JPA API

Get a single employee

JPA API

Get a single employee

JPA API

```
@Override
public Employee findById(int theId) {
```

Get a single employee

JPA API

```
@Override
public Employee findById(int theId) {

  // get employee
  Employee theEmployee = entityManager.find(Employee.class, theId);
```

Get a single employee

JPA API

```
@Override
public Employee findById(int theId) {

  // get employee
  Employee theEmployee = entityManager.find(Employee.class, theId);

  // return employee
  return theEmployee;
}
```

Add or Update employee

JPA API

Add or Update employee

JPA API

```
@Override
public void save(Employee theEmployee) {
```

Add or Update employee

JPA API

```
@Override
public void save(Employee theEmployee) {

  // save or update the employee
  Employee dbEmployee = entityManager.merge(theEmployee);
```

Add or Update employee

JPA API

```
@Override
public void save(Employee theEmployee) {

  // save or update the employee
  Employee dbEmployee = entityManager.merge(theEmployee);
```

if id == 0
then save/insert
else update

Add or Update employee

JPA API

if id == 0
then save/insert
else update

```
@Override
public void save(Employee theEmployee) {

  // save or update the employee
  Employee dbEmployee = entityManager.merge(theEmployee);

  // update with id from db ... so we can get generated id for save/insert
  theEmployee.setId(dbEmployee.getId());
}
```

Add or Update employee

JPA API

```
@Override
public void save(Employee theEmployee) {

  // save or update the employee
  Employee dbEmployee = entityManager.merge(theEmployee);

  // update with id from db ... so we can get generated id for save/insert
  theEmployee.setId(dbEmployee.getId());
}
```

if id == 0
then save/insert
else update

Useful in our REST API
to return generated id

Delete an existing employee

JPA API

Delete an existing employee

JPA API

```
@Override
public void deleteById(int theId) {
```

Delete an existing employee

JPA API

```
@Override
public void deleteById(int theId) {

  // delete object with primary key
  Query theQuery = entityManager.createQuery(
                                  "delete from Employee where id=:employeeId");
```

Delete an existing employee

JPA API

```
@Override
public void deleteById(int theId) {

  // delete object with primary key
  Query theQuery = entityManager.createQuery(
                                 "delete from Employee where id=:employeeId");

  theQuery.setParameter("employeeId", theId);
```

Delete an existing employee

JPA API

```
@Override
public void deleteById(int theId) {

  // delete object with primary key
  Query theQuery = entityManager.createQuery(
                                 "delete from Employee where id=:employeeId");

  theQuery.setParameter("employeeId", theId);
```

Delete an existing employee

JPA API

```
@Override
public void deleteById(int theId) {

  // delete object with primary key
  Query theQuery = entityManager.createQuery(
                                  "delete from Employee where id=:employeeId");

  theQuery.setParameter("employeeId", theId);

  theQuery.executeUpdate();
}
```