Spring Boot REST API - Real Time Project

Real-Time Project

REST API with Spring Boot that connects to a database

API Requirements

API Requirements

From the Boss

Create a REST API for the Employee Directory

API Requirements

From the Boss

Create a REST API for the Employee Directory

REST clients should be able to

API Requirements

From the Boss

Create a REST API for the Employee Directory

REST clients should be able to

- Get a list of employees

API Requirements

From the Boss

Create a REST API for the Employee Directory

REST clients should be able to

- Get a list of employees
- Get a single employee by id

API Requirements

From the Boss

Create a REST API for the Employee Directory

REST clients should be able to

- Get a list of employees
- Get a single employee by id
- Add a new employee

API Requirements

From the Boss

Create a REST API for the Employee Directory

REST clients should be able to

- Get a list of employees
- Get a single employee by id
- Add a new employee
- Update an employee

API Requirements

From the Boss

Create a REST API for the Employee Directory

REST clients should be able to

- Get a list of employees
- Get a single employee by id
- Add a new employee
- Update an employee
- Delete an employee

REST API

REST API

HTTP Method		CRUD Action
POST	/api/employees	Create a new employee

REST API

HTTP Method		CRUD Action
POST	/api/employees	Create a new employee
GET	/api/employees	Read a list of employees

REST API

HTTP Method		CRUD Action
POST	/api/employees	Create a new employee
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee

REST API

HTTP Method		CRUD Action
POST	/api/employees	Create a new employee
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
PUT	/api/employees	Update an existing employee

REST API

HTTP Method		CRUD Action
POST	/api/employees	Create a new employee
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Development Process

Step-By-Step

Development Process

Step-By-Step

1. Set up Database Dev Environment

Development Process

Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr

Development Process

Step-By-Step

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees

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

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

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

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

Application Architecture

Application Architecture

Employee
REST
Controller

Application Architecture

Application Architecture

Application Architecture

Application Architecture

Spring Boot
All Java Config
No XML

Setup Database Table

employee.sql

employee.sql

1. File: employee.sql

employee.sql

1. File: employee.sql

2. Create a new database table: **employee**

employee.sql

1. File: employee.sql

2. Create a new database table: **employee**

3. Load table with sample data

employee.sql

1. File: employee.sql

2. Create a new database table: **employee**

3. Load table with sample data

Details on downloading the SQL file provided in next section

Creating Spring Boot Project

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

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

Our Task

Spring Initializer

Create DAO in Spring Boot

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

Application Architecture

Application Architecture

Hibernate Session Factory

Hibernate Session Factory

- In the past, our DAO used a Hibernate Session Factory

Hibernate Session Factory

- In the past, our DAO used a Hibernate Session Factory
- Hibernate Session Factory needs a Data Source

Hibernate Session Factory

- In the past, our DAO used a Hibernate Session Factory
- Hibernate Session Factory needs a Data Source
  - The data source defines database connection info

Traditional Spring

Traditional Spring

- We normally had to do this configuration manually

Traditional Spring

- We normally had to do this configuration manually

XML

```
<!-- Step 1: Define Database DataSource / connection pool -->
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/web_customer_tracker?
    <property name="user" value="springstudent" />
    <property name="password" value="springstudent" />

    <!-- these are connection pool properties for C3P0 -->
    <property name="minPoolSize" value="5" />
    <property name="maxPoolSize" value="20" />
    <property name="maxIdleTime" value="30000" />
</bean>

<!-- Step 2: Setup Hibernate session factory -->
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />
    <property name="packagesToScan" value="com.luv2code.springdemo.entity" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<!-- Step 3: Setup Hibernate transaction manager -->
<bean id="myTransactionManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Traditional Spring

- We normally had to do this configuration manually

XML

```xml
<!-- Step 1: Define Database DataSource / connection pool -->
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/web_customer_tracker?
    <property name="user" value="springstudent" />
    <property name="password" value="springstudent" />

    <!-- these are connection pool properties for C3P0 -->
    <property name="minPoolSize" value="5" />
    <property name="maxPoolSize" value="20" />
    <property name="maxIdleTime" value="30000" />
</bean>

<!-- Step 2: Setup Hibernate session factory -->
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />
    <property name="packagesToScan" value="com.luv2code.springdemo.entity" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<!-- Step 3: Setup Hibernate transaction manager -->
<bean id="myTransactionManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

ALL JAVA

```java
@Bean
public DataSource myDataSource() {
    ComboPooledDataSource myDataSource = new ComboPooledDataSource();
    try {
        myDataSource.setDriverClass("com.mysql.jdbc.Driver");
    }
    catch (PropertyVetoException exc) {
        throw new RuntimeException(exc);
    }

    myDataSource.setJdbcUrl(env.getProperty("jdbc.url"));
    myDataSource.setUser(env.getProperty("jdbc.user"));
    myDataSource.setPassword(env.getProperty("jdbc.password"));
    myDataSource.setInitialPoolSize(getIntProperty("connection.pool.initialPoolSize"));
    myDataSource.setMinPoolSize(getIntProperty("connection.pool.minPoolSize"));
    myDataSource.setMaxPoolSize(getIntProperty("connection.pool.maxPoolSize"));
    myDataSource.setMaxIdleTime(getIntProperty("connection.pool.maxIdleTime"));

    return myDataSource;
}

@Bean
public LocalSessionFactoryBean sessionFactory(){

    // create session factorys
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();

    // set the properties
    sessionFactory.setDataSource(myDataSource());
    sessionFactory.setPackagesToScan(env.getProperty("hibernate.packagesToScan"));
    sessionFactory.setHibernateProperties(getHibernateProperties());

    return sessionFactory;
}
```

Traditional Spring

- We normally had to do this configuration manually

Very error-prone

Easy to make a simple mistake

XML

```xml
<!-- Step 1: Define Database DataSource / connection pool -->
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/web_customer_tracker?
    <property name="user" value="springstudent" />
    <property name="password" value="springstudent" />

    <!-- these are connection pool properties for C3P0 -->
    <property name="minPoolSize" value="5" />
    <property name="maxPoolSize" value="20" />
    <property name="maxIdleTime" value="30000" />
</bean>

<!-- Step 2: Setup Hibernate session factory -->
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />
    <property name="packagesToScan" value="com.luv2code.springdemo.entity" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<!-- Step 3: Setup Hibernate transaction manager -->
<bean id="myTransactionManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

ALL JAVA

```java
@Bean
public DataSource myDataSource() {
    ComboPooledDataSource myDataSource = new ComboPooledDataSource();
    try {
        myDataSource.setDriverClass("com.mysql.jdbc.Driver");
    }
    catch (PropertyVetoException exc) {
        throw new RuntimeException(exc);
    }

    myDataSource.setJdbcUrl(env.getProperty("jdbc.url"));
    myDataSource.setUser(env.getProperty("jdbc.user"));
    myDataSource.setPassword(env.getProperty("jdbc.password"));
    myDataSource.setInitialPoolSize(getIntProperty("connection.pool.initialPoolSize"));
    myDataSource.setMinPoolSize(getIntProperty("connection.pool.minPoolSize"));
    myDataSource.setMaxPoolSize(getIntProperty("connection.pool.maxPoolSize"));
    myDataSource.setMaxIdleTime(getIntProperty("connection.pool.maxIdleTime"));

    return myDataSource;
}

@Bean
public LocalSessionFactoryBean sessionFactory(){

    // create session factorys
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();

    // set the properties
    sessionFactory.setDataSource(myDataSource());
    sessionFactory.setPackagesToScan(env.getProperty("hibernate.packagesToScan"));
    sessionFactory.setHibernateProperties(getHibernateProperties());

    return sessionFactory;
}
```

Traditional Spring

- We normally had to do this configuration manually

Very error-prone

Easy to make a simple mistake

XML

```xml
<!-- Step 1: Define Database DataSource / connection pool -->
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
```

```xml
</bean>
```

ALL JAVA

```java
@Bean
public DataSource myDataSource() {
    ComboPooledDataSource myDataSource = new ComboPooledDataSource();
    try {
        myDataSource.setDriverClass("com.mysql.jdbc.Driver");
```

```
}
```

There should be an easier solution

Spring Boot to the Rescue

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you

- Based on entries from Maven pom file

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you

- Based on entries from Maven pom file
  - JDBC Driver: **`mysql-connector-java`**

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you

- Based on entries from Maven pom file
  - JDBC Driver: **`mysql-connector-java`**
  - Spring Data (ORM): **`spring-boot-starter-data-jpa`**

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you

- Based on entries from Maven pom file
  - JDBC Driver: **`mysql-connector-java`**
  - Spring Data (ORM): **`spring-boot-starter-data-jpa`**

- DB connection info from **`application.properties`**

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory
spring.datasource.username=springstudent
spring.datasource.password=springstudent
```

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory
spring.datasource.username=springstudent
spring.datasource.password=springstudent
```

No need to give JDBC driver class name
Spring Boot will automatically detect it based on URL

Additional Data Source Properties

Additional Data Source Properties

- Properties are available to configure connection pool etc

Additional Data Source Properties

- Properties are available to configure connection pool etc

List of Data Source Properties

www.luv2code.com/spring-boot-props

Additional Data Source Properties

- Properties are available to configure connection pool etc

List of Data Source Properties

www.luv2code.com/spring-boot-props

```
spring.datasource.*
```

Auto Data Source Configuration

Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:

Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
  - **`DataSource, EntityManager,`** …

Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
  - **`DataSource, EntityManager,`** …

- You can then inject these into your app, for example your DAO

Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
  - **`DataSource, EntityManager,`** …
- You can then inject these into your app, for example your DAO
- **`EntityManager`** is from Java Persistence API (JPA)

What is JPA?

What is JPA?

- Java Persistence API (JPA)

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification
  - Defines a set of interfaces

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification
  - Defines a set of interfaces
  - Requires an implementation to be usable

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification
  - Defines a set of interfaces
  - Requires an implementation to be usable

www.luv2code.com/jpa-spec

JPA - Vendor Implementations

JPA - Vendor Implementations

JPA Spec

JPA - Vendor Implementations

JPA - Vendor Implementations

JPA - Vendor Implementations

www.luv2code.com/jpa-vendors

What are Benefits of JPA

What are Benefits of JPA

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

What are Benefits of JPA

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

- Maintain portable, flexible code by coding to JPA spec (interfaces)

What are Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations

What are Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations
  - For example, if Vendor ABC stops supporting their product

What are Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations
  - For example, if Vendor ABC stops supporting their product
  - You could switch to Vendor XYZ without vendor lock in

JPA - Vendor Implementations

JPA - Vendor Implementations

My Biz
App

JPA - Vendor Implementations

JPA - Vendor Implementations

JPA - Vendor Implementations

Auto Data Source Configuration

Auto Data Source Configuration

- In Spring Boot, Hibernate is default implementation of JPA

Auto Data Source Configuration

- In Spring Boot, Hibernate is default implementation of JPA

- **`EntityManager`** is similar to Hibernate **`SessionFactory`**

Auto Data Source Configuration

- In Spring Boot, Hibernate is default implementation of JPA
- **`EntityManager`** is similar to Hibernate **`SessionFactory`**
- **`EntityManager`** can serve as a wrapper for a Hibernate **`Session`** object

Auto Data Source Configuration

- In Spring Boot, Hibernate is default implementation of JPA
- **`EntityManager`** is similar to Hibernate **`SessionFactory`**
- **`EntityManager`** can serve as a wrapper for a Hibernate **`Session`** object
- We can inject the **`EntityManager`** into our DAO

Show Me The Code!

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

DAO Interface

```
public interface EmployeeDAO {

 public List<Employee> findAll();

}
```

DAO Impl

DAO Impl

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

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;
```

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;

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

 ...
}
```

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;

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

 ...
}
```

Automatically created by Spring Boot

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;

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

 ...
}
```

Automatically created by Spring Boot

Constructor injection

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;

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

 ...
}
```

Optional

Automatically created by Spring Boot

Constructor injection

DAO Impl

DAO Impl

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

DAO Impl

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

  // get the current hibernate session
  Session currentSession = entityManager.unwrap(Session.class);
```

Get current
Hibernate session

DAO Impl

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

  // get the current hibernate session
  Session currentSession = entityManager.unwrap(Session.class);

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

Get current Hibernate session

DAO Impl

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

  // get the current hibernate session
  Session currentSession = entityManager.unwrap(Session.class);

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

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

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

Get current Hibernate session

Using native Hibernate API

Development Process

Step-By-Step

Development Process

Step-By-Step

1. Update db configs in application.properties

Development Process

Step-By-Step

1. Update db configs in application.properties
2. Create Employee entity

Development Process

Step-By-Step

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface

Development Process

Step-By-Step

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation

Development Process

Step-By-Step

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation
5. Create REST controller to use DAO

DAO: Find, Add, Update and Delete

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

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

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

DAO methods

Service: Find, Add, Update and Delete

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

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

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

Service methods

Rest Controller Methods - Find and Add

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

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

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

Rest Controller methods

Real-Time Project

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
✓ GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Read a Single Employee

Read a Single Employee

REST
Client

Read a Single Employee

REST Client

Employee REST Controller

Read a Single Employee

Read a Single Employee

REST Client

GET /api/employees/{employeeId}

```
{
    "id": 1,
    "firstName": "David",
    "lastName": "Adams",
    "email": "david@luv2code.com"
}
```

Employee REST Controller

Create a New Employee

REST Client

Employee REST Controller

Create a New Employee

REST Client

POST /api/employees

```
{
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

Employee REST Controller

Create a New Employee

Since new employee, we are not passing id / primary key

REST Client

POST /api/employees

```
{
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

Employee REST Controller

Create a New Employee

Since new employee, we are not passing id / primary key

REST Client

POST /api/employees

```
{
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

```
{
    "id": 7,
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

Employee REST Controller

Create a New Employee

Since new employee, we are not passing id / primary key

REST Client

POST /api/employees

```
{
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

Employee REST Controller

```
{
    "id": 7,
    "firstName": "Juan",
    "lastName": "Perez",
    "email": "juan.perez@luv2code.com"
}
```

Response contains new id / primary key

Sending JSON to Spring REST Controllers

Sending JSON to Spring REST Controllers

- When sending JSON data to Spring REST Controllers

Sending JSON to Spring REST Controllers

- When sending JSON data to Spring REST Controllers
- For controller to process JSON data, need to set a HTTP request header

Sending JSON to Spring REST Controllers

- When sending JSON data to Spring REST Controllers

- For controller to process JSON data, need to set a HTTP request header
  - `Content-type: application/json`

Sending JSON to Spring REST Controllers

- When sending JSON data to Spring REST Controllers
- For controller to process JSON data, need to set a HTTP request header
  - `Content-type: application/json`
- Need to configure REST client to send the correct HTTP request header

Postman - Sending JSON in Request Body

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman

Postman - Sending JSON in Request Body

- Must set HTTP request header in Postman

Based on these configs, Postman will automatically set the correct HTTP request header

Rest Controller Methods - Update and Delete

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

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

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

Rest Controller methods

Real-Time Project

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
✓ GET	/api/employees	Read a list of employees
✓ GET	/api/employees/{employeeId}	Read a single employee
✓ POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Real-Time Project

Checkpoint

HTTP Method		CRUD Action
GET	/api/employees	Read a list of employees
GET	/api/employees/{employeeId}	Read a single employee
POST	/api/employees	Create a new employee
PUT	/api/employees	Update an existing employee
DELETE	/api/employees/{employeeId}	Delete an existing employee

Update Employee

REST Client

Employee REST Controller

Update Employee

REST Client

PUT /api/employees

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

Employee REST Controller

Update Employee

ID of employee to update
With updated info

REST Client

PUT /api/employees

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

Employee REST Controller

Update Employee

ID of employee to update
With updated info

REST Client

PUT /api/employees

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

Employee REST Controller

Update Employee

ID of employee to update
With updated info

REST Client

PUT /api/employees

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

Employee REST Controller

```
{
    "id" : 1,
    "firstName" : "Daniel",
    "lastName" : "Vega",
    "email" : "daniel.vega@luv2code.com"
}
```

Response contains updated info (echoed)

Delete Employee

REST Client

Employee REST Controller

Delete Employee

Delete Employee