Integrating with Data Sources

1. Essential concepts
2. Getting started with JPA
3. Defining an entity class
4. Managing entities
5. Using Spring Data repositories

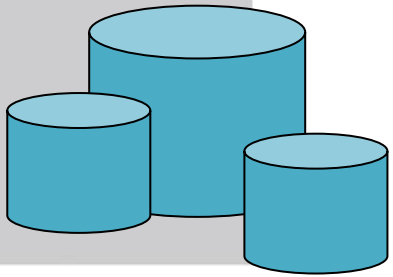
Demo Spring module: demo-spring-data

1. Essential Concepts

- Spring vertical data access APIs
- Spring Data project
- Configuring Maven dependencies

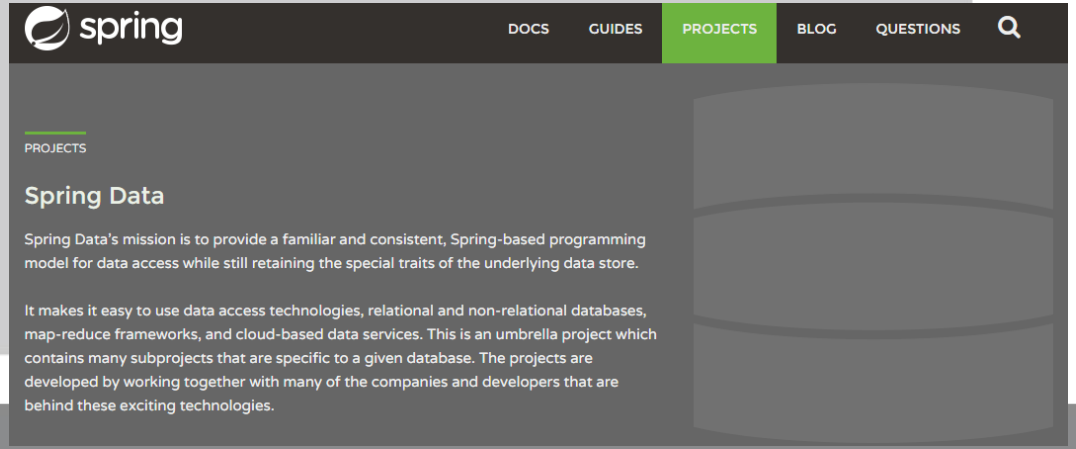
Spring Vertical Data Access APIs

- Spring provides vertical APIs for data access
  - Many technologies, including JDBC, JPA, Hibernate, etc.
- Declarative transaction management
  - Transactional boundaries declared via configuration
  - Enforced by a Spring transaction manager
- Automatic connection management
  - Acquires/releases connections automatically



Spring Data Project

- In recent times, the Spring Data project has emerged
  - Supports a wider range of data access technologies, including REST, RDBMS, NoSQL, elastic search, etc.
  - Powerful repository and object-mapping abstractions
  - Dynamic query creation from repository method names



Configuring Maven Dependencies

- Add the appropriate Maven dependency for the type of data source you wish to access
- We'll use an H2 in-memory database in our demos
  - Database is created/dropped when app starts/ends

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

pom.xml

- Spring Boot does a lot of auto-configuration, based on the data sources it sees in your pom file (see later)

2. Getting Started with JPA

- Overview of JPA
- Important JPA concepts
- JPA dependency in Spring Boot
- Spring Boot autoconfiguration
- Customizing persistence properties

Demo package: `demo.springdata.jpa`

Overview of JPA

- JPA = Java Persistence API
  - A standard ORM (object/relational mapping) API
- JPA is a specification
  - Implemented by the Hibernate library
  - Also implemented by Java Enterprise Edition
- To use JPA in Spring:
  - Add the Hibernate library to your classpath (see later)

Important JPA Concepts

- Entity class - maps a class to a db table
  - Entity objects correspond to rows in the db table
- Entity manager - enables you to fetch entities from db
  - Also automatically flushes modified entities to the db
- Entity manager factory - creates an entity manager
  - Configures the entity manager so it can connect to a db

JPA Dependency in Spring Boot

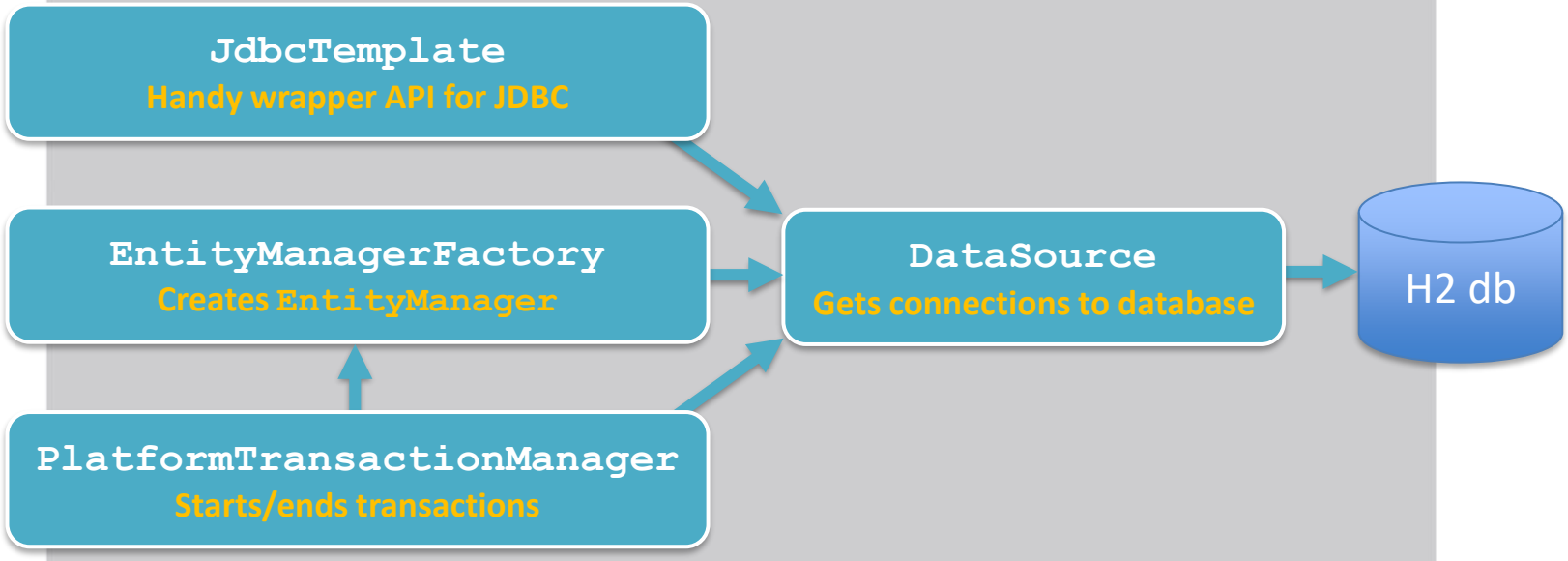
- To use JPA in a Spring Boot app, you need to add the following dependency to your pom file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

pom.xml

Spring Boot Autoconfiguration

- Courtesy of the JPA dependency, Spring Boot creates several beans automatically in your application



Customizing Persistence Properties

- Spring Boot automatically sets persistence properties to connect to the in-memory H2 database:

```
spring.datasource.url=jdbc:h2:mem:example
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

- You can customize persistence properties if you need to

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true
spring.jpa.properties.hibernate.format_sql=true
```

application.properties

3. Defining an Entity Class

- How to define an entity class
- Locating entity classes
- Seeding the database with data
- Viewing the database data

How to Define an Entity Class

- You can define an entity class as follows:

```
import javax.persistence.*;

@Entity
@Table(name="EMPLOYEES")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long employeeId = -1;

    private String name;
    private String region;

    @Column(name="salary")
    private double dosh;

    // Plus constructors, getters/setters, equals(), hashCode(), and toString()
}
```

Employee.java

Locating Entity Classes

- A Spring Boot app scans for entity classes when it starts
  - It looks in the main app class package, plus sub-packages
- You can tell it to look elsewhere, if you like
  - Via `@EntityScan`

```
@SpringBootApplication
@EntityScan( {"myentitypackage1", "myentitypackage2"} )
public class Application {
    …
}
```

Seeding the Database with Data

- For convenience during development/testing, you can seed the database with some sample data

```
import org.springframework.jdbc.core.JdbcTemplate;
…

@Component
public class SeedDb {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        jdbcTemplate.update("insert into EMPLOYEES(name,salary,region) values(?,?,?)",
                            new Object[]{"James", 21000, "London"});
        …
    }
}
```

SeedDb.java

Viewing the Database Data (1 of 3)

- Most databases have a console UI, to let you view data
  - To enable the H2 console UI, add these app properties:

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

application.properties

- The H2 console UI is a web endpoint, so you must also add the Spring Boot web dependency to your pom:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

pom.xml

Viewing the Database Data (2 of 3)

- When you run your app, you'll see a message that indicates the connection URL for the database

```
.kariDataSource       : HikariPool-1 - Start completed.
:AutoConfiguration    : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:d58eb18c-b573-4967-a6e2-ce52b628e561'
ernal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
on                    : HHH000412: Hibernate ORM core version 5.4.28.Final
.ons.common.Version   : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
```

- You can use this URL to connect to the database in the H2 console UI - see next slide

Viewing the Database Data (3 of 3)

- To open the H2 console UI, browse to:
  - http://localhost:8080/h2-console
- To connect to the database, enter these details:
  - JDBC URL - as per previous slide
  - User name - `sa`
  - Password - leave blank
- You can then view tables etc. in the database - cool!

4. Managing Entities

- Defining a repository class
- Performing a simple query
- Finding an entity by primary key
- Getting a list of entities
- Performing data modification operations

Defining a Repository Class

- In the demo, we put our JPA code in a repository class
  - We use an injected `EntityManager` to do the work

```
import javax.persistence.*;
…

@Repository
public class EmployeeRepository {

    @PersistenceContext
    private EntityManager entityManager;

    // Methods to create, read, update, and delete database records.
    // See following slides for details…

}
```

EmployeeRepository.java

Performing a Simple Query

- Define a query string
  - Using JPQL (or SQL)
- Create a `TypedQuery<T>` object
  - Via `createQuery()` on the `EntityManager`
- Execute the query, and get a single result back
  - Via `getSingleResult()` on the query object

```
public long getEmployeeCount() {
    String jpql = "select count(e) from Employee e";
    TypedQuery<Long> query = entityManager.createQuery(jpql, Long.class);
    return query.getSingleResult();
}
```

EmployeeRepository.java

Finding an Entity by Primary Key

- To find an entity by primary key:
  - Call `find()` on the `EntityManager`
  - Returns `null` if entity not found

```
public Employee getEmployee(long employeeId) {
    return entityManager.find(Employee.class, employeeId);
}
```

EmployeeRepository.java

Getting a List of Entities

- To get a list of entities:
  - Call `getResultList()` on a query object

```
public List<Employee> getEmployees() {
    String jpql = "select e from Employee e";
    TypedQuery<Employee> query = entityManager.createQuery(jpql, Employee.class);
    return query.getResultList();
}
```

EmployeeRepository.java

Performing Data Modification Operations

- This is how you insert, update, and delete entities using JPA - also note the need for `@Transactional`

```
@Transactional
public void insertEmployee(Employee e) {
    entityManager.persist(e);
}

@Transactional
public void updateEmployee(Employee e) {
    Employee entity = entityManager.find(Employee.class, e.getEmployeeId());
    entity.setName(e.getName());
    entity.setDosh(e.getDosh());
    entity.setRegion(e.getRegion());
}

@Transactional
public void deleteEmployee(long employeeId) {
    Employee e = entityManager.find(Employee.class, employeeId);
    entityManager.remove(e);
}
```

EmployeeRepository.java

5. Using Spring Data Repositories

- Overview
- Spring Data repository capabilities
- Domain-specific repositories
- Locating Spring Data repositories
- Using Spring Data repositories

Demo package: `demo.springdata.repos`

Overview

- Spring Data is a data-access abstraction mechanism
  - Makes it very easy to access a wide range of data stores
  - Using a familiar "repository" pattern
- It provides template repositories for…
  - JPA
  - MongoDB, Cassandra, Neo4J, DynamoDB, etc.
  - Etc.

Spring Data Repository Capabilities

- Spring Data defines agnostic data-access repository interfaces, e.g. `CrudRepository`

`long`	`count()` Returns the number of entities available.
`void`	`delete(ID id)` Deletes the entity with the given id.
`void`	`delete(Iterable<? extends T> entities)` Deletes the given entities.
`void`	`delete(T entity)` Deletes a given entity.
`void`	`deleteAll()` Deletes all entities managed by the repository.
`boolean`	`exists(ID id)` Returns whether an entity with the given id exists.
`Iterable<T>`	`findAll()` Returns all instances of the type.
`Iterable<T>`	`findAll(Iterable<ID> ids)` Returns all instances of the type with the given IDs.
`T`	`findOne(ID id)` Retrieves an entity by its id.
`<S extends T> Iterable<S>`	`save(Iterable<S> entities)` Saves all given entities.
`<S extends T> S`	`save(S entity)` Saves a given entity.

Domain-Specific Repositories (1 of 2)

- You can define your own domain-specific interfaces
  - Extend `CrudRepository`
  - Specify the entity type and the PK type
- You can define specific query methods for your entities
  - Spring Data reflects on method names to create queries
  - You can provide explicit JPQL syntax for complex queries

For details about Spring Data repositories, see:
https://docs.spring.io/spring-data/data-commons/docs/2.4.x/reference/html/#repositories

Domain-Specific Repositories (2 of 2)

- Here's an example of a domain-specific repository
  - Entity type is `Employee`, PK type is `Long`
  - Also we've defined some additional queries

```
public interface EmployeeRepository extends CrudRepository<Employee, Long> {

    List<Employee> findByRegion(String region);

    @Query("select emp from Employee emp where emp.dosh >= ?1 and emp.dosh <= ?2")
    List<Employee> findInSalaryRange(double from, double to);

    Page<Employee> findByDoshGreaterThan(double salary, Pageable pageable);

}
```

EmployeeRepository.java

Locating Spring Data Repositories

- A Spring Boot app scans for Spring Data JPA repository interfaces when it starts
  - It looks in the main app class package, plus subpackages
- You can tell it to look elsewhere, if you like

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
…

@SpringBootApplication
@EnableJpaRepositories({"repopackage1", "repopackage2"})
public class Application {
    …
}
```

Using Spring Data Repositories

```
@Component
public class EmployeeService {

   @Autowired
   private EmployeeRepository repository;

   public void doDemo() {

      // Insert an employee.
      Employee newEmp = new Employee(-1, "Simon Peter", 10000, "Israel");
      repository.save(newEmp);
      System.out.printf("There are now %d employees\n", repository.count());

      // Get all employees.
      displayEmployees("All employees after insert: ", repository.findAll());

      // Get employees by salary range.
      List<Employee> emps = repository.findInSalaryRange(20000, 50000);
      displayEmployees("Employees earning 20k to 50k: ", emps);

      // Get a page of employees.
      Pageable pageable = PageRequest.of(1, 3, Direction.DESC, "dosh");
      Page<Employee> page = repository.findByDoshGreaterThan(50000, pageable);
      displayEmployees("Page 1 of employees more than 50k: ", page.getContent());
   }
   …
```

EmployeeService.java

Summary

- Essential concepts
- Getting started with JPA
- Defining an entity class
- Managing entities
- Using Spring Data repositories