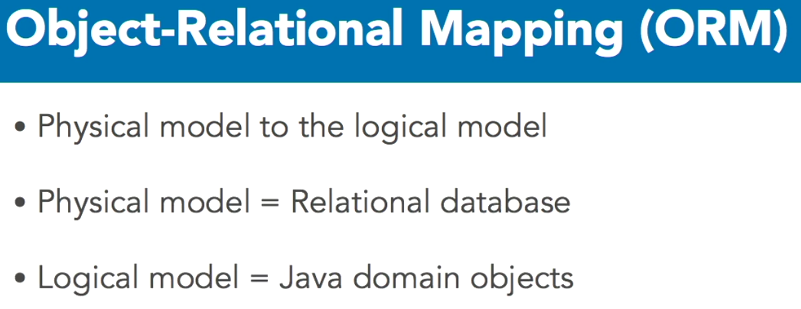


The repository pattern is the abstraction used by Spring Data commons to accomplish these goals.

It is followed throughout the Spring Data project for creating, reading, updating, and deleting records by citing entities.

Any module for a data source has a repository that extends from the generic one. So, for example, Spring Data JPA has a JPA repository, Spring Data mongo DB has a mongo repository, Spring Data gemfire has a gemfire repository, and so on.



Typical physical models are relational database repository, and a logical model is the Java domain objects. It's possible to do ORM in plain vanilla Java, but it's nasty.

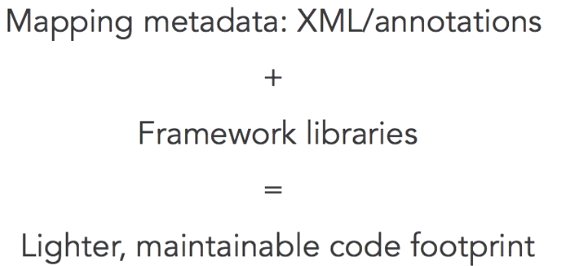
First you must

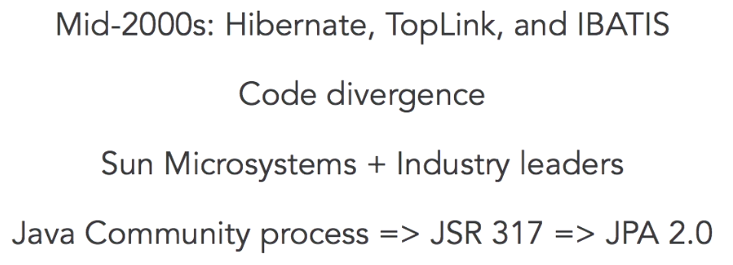
1. open a transaction,
2. query the database which returns the records
3. for each record iterate through the fields
4. for each field extract the data invoking the proper type conversion
5. map the data to the Java object or object attribute and
6. close the transaction.
7. Queries that modify the database require committing the transaction or rolling it back if an error occurs.

Programming this way is tedious. The code is difficult to refactor and maintain.

Object to relational mapping frameworks solve this problem. With an ORM framework a developer declares metadata to associate Java classes and attributes to database tables and columns.

The framework then handles the actual database interactions. So, the code is abstracted away from the database. This makes it lighter and easier to maintain.

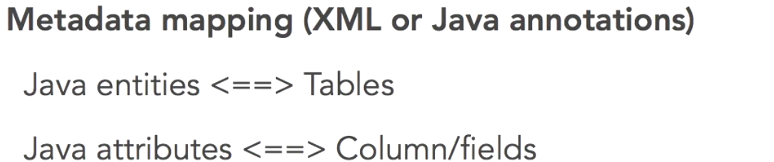


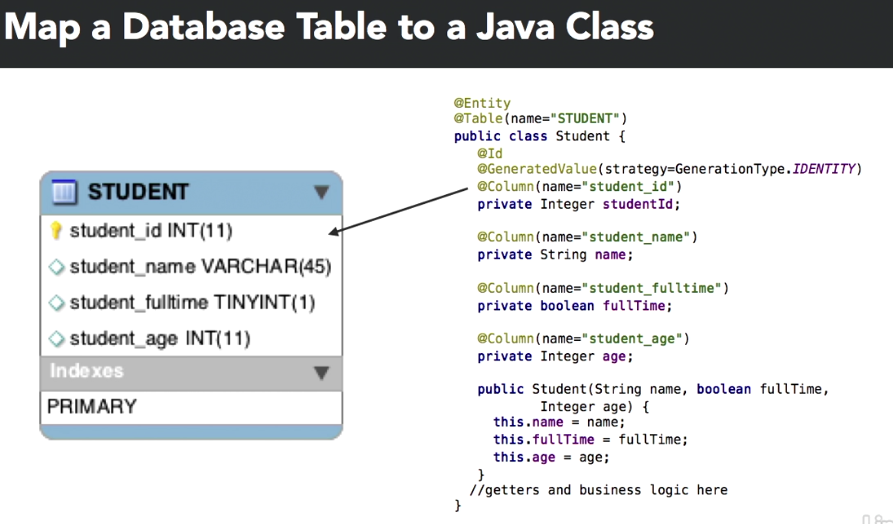


The most popular ORM framework is JPA. So how did that happen? What's the backstory of JPA? In the early to mid 2000'S, ORM frameworks arrived. The most popular were Hibernate, TopLink, and IBATIS. Different frameworks diverge development so industry experts work together to form an ORM standard. The result was the Java persistence application programming interface. Through the Java Community process, the JPA spec version 2.0 final a.k.a. JSR 317 was released in December 2009.

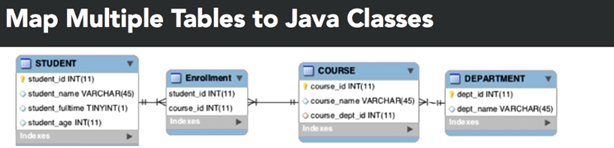
**JPA is a specification, it is not an actual framework**. Framework, such as Hibernate and EclipseLink, as well as Java Enterprise Edition compliant application servers provide the implementation. So even if we speak about a Java application that uses JPA, it is using another provider following the JPA spec. The spec includes metadata declarations to map Java classes, now called entities, to database tables.

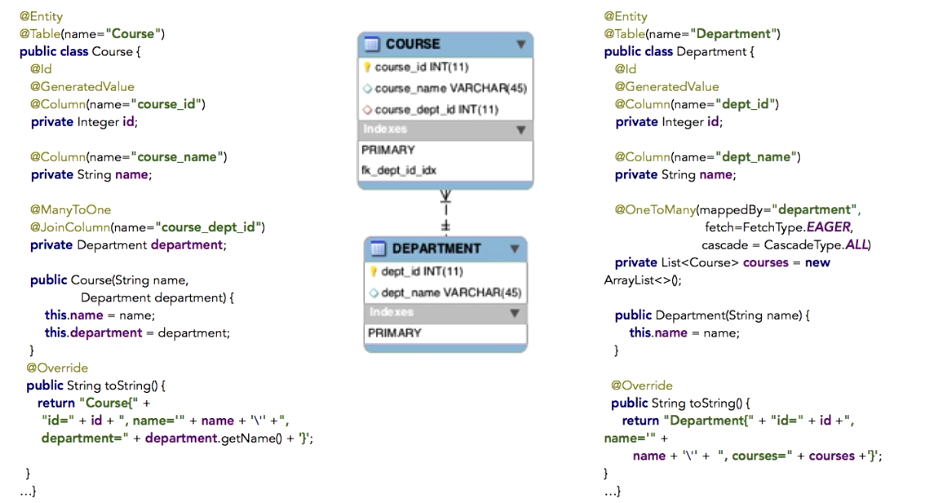
An EntityManager creates, reads, updates, and deletes the entities. Changes to the entity's state are reflected to the database. The entity metadata is in the form of .xml files, or within Java classes, via Java annotations. This tutorial only demonstrates Java annotations. Please refer to the JPA specification if you prefer xml.





Now that we have identified the object to relational mapping metadata, all of our database related coding can stay in the logical world because JPA will take care of the physical world for us.

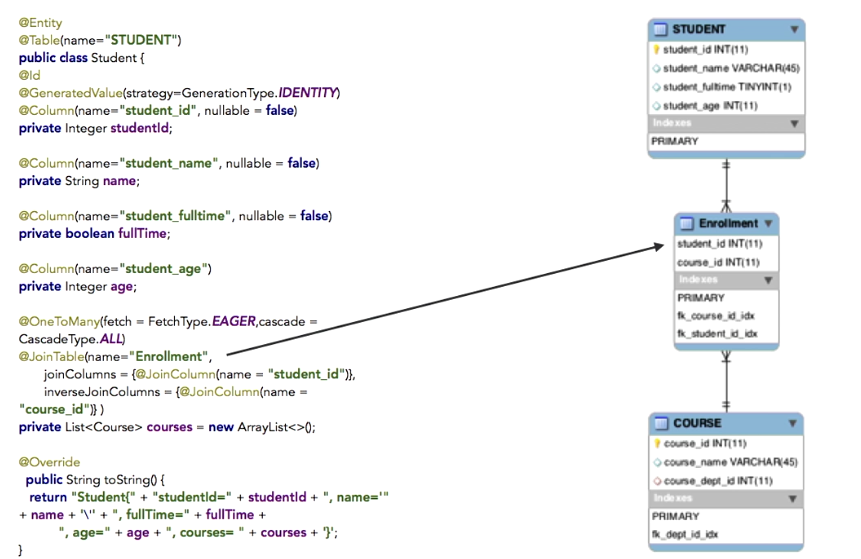


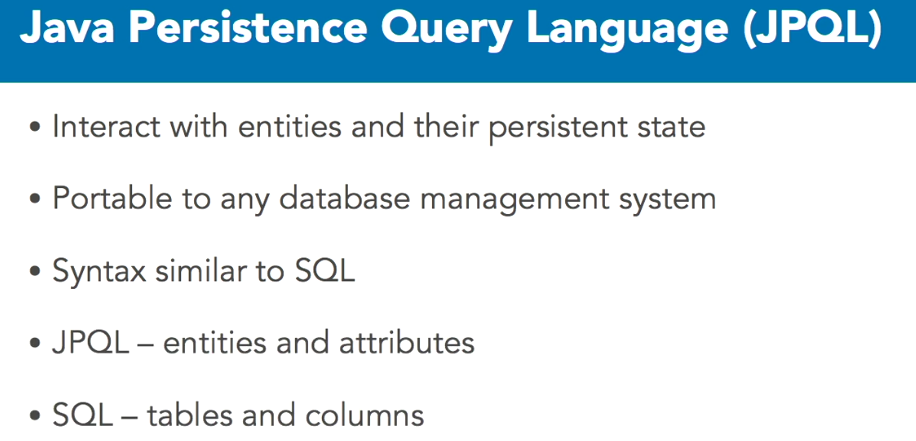


The course table has a course underscore department underscore ID column. The **@join** annotation declares this physical mapping enabling a logical relationship between the course entity and the department entity. The **@many to one** annotation shows cardinality. Many courses are mapped to one department. Now let's look at the other side of the relationship. Department to courses. The department entity has a list of courses that annotates with @one to many.

One department has many courses. The mapped by equals department parameter refers to the department attribute back in the course class. We also see fetch and cascade parameters in the one to many annotations. By default, one to many attributes are not automatically fetched from the database. This is called lazy loading, but we can override that with the fetchtype.eager setting. So, when a department is read from the database, JPA also populates the associated courses.

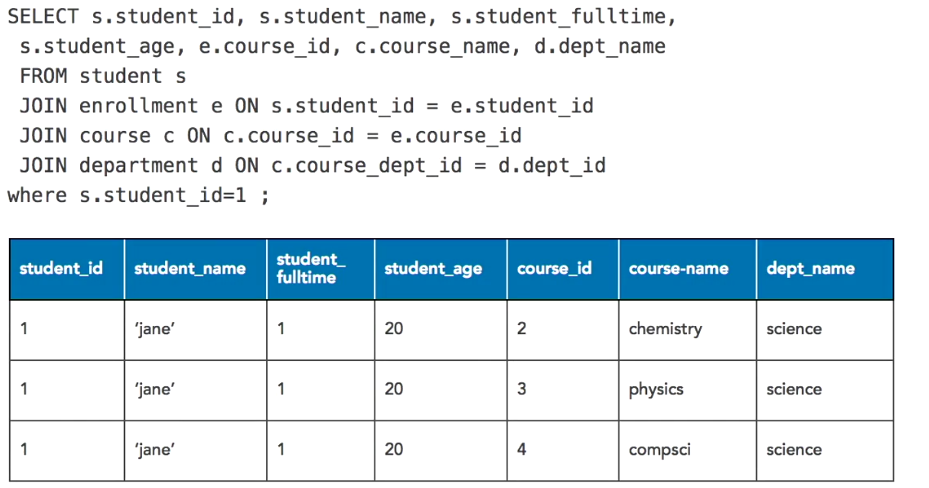
Cascade controls the state of the collection attributes for an entity. With cascade set to all, any state changes that are made in the department should apply to its courses as well



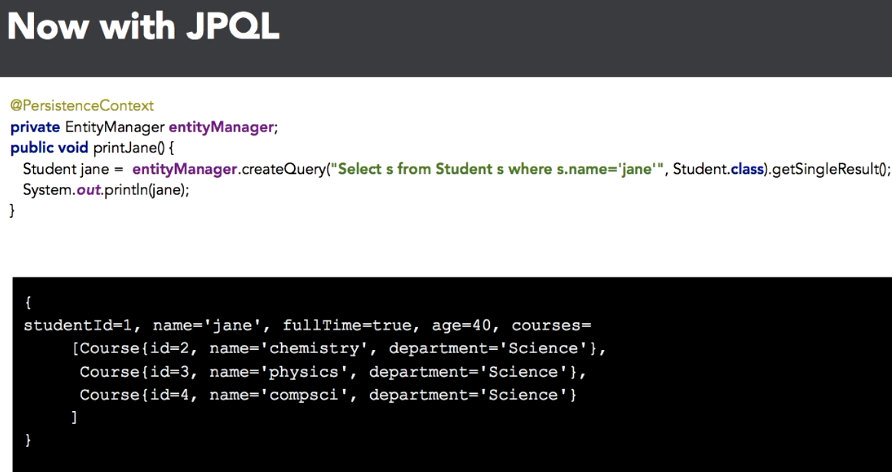


**Without JPQL:**

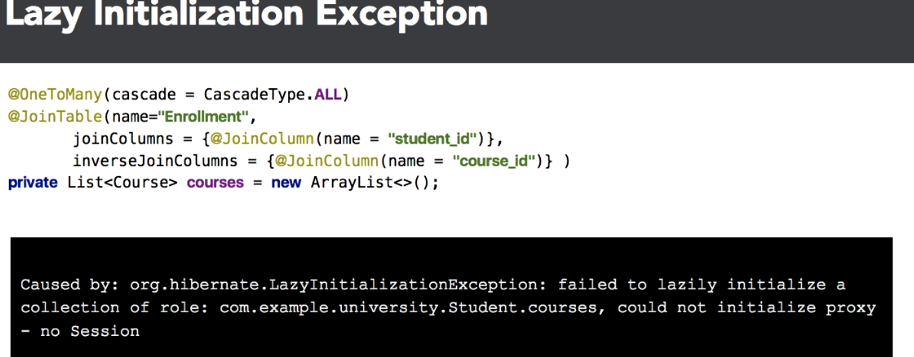
The below SQL query fetches Jane's details as well as the courses she's taking. It accesses all four tables to pull in seven columns of information. The first four are from the student table. Course\_id is from the enrollment table. Course-name is from the course table and dept\_name is from the department table. This query is messy and difficult to maintains and we still need a lot of coding to map it from the physical model to the logical model.



**With JPQL:**



a method called print jane which invokes entityManager.createQuery passing in the JPQL query, Select s from Student where s.name=jane. Get single result invokes the query and returns a student object, then we print out the student to the console. Not only does the query pull the student object, it also grabs all the courses because we set the fetch type equals eager.



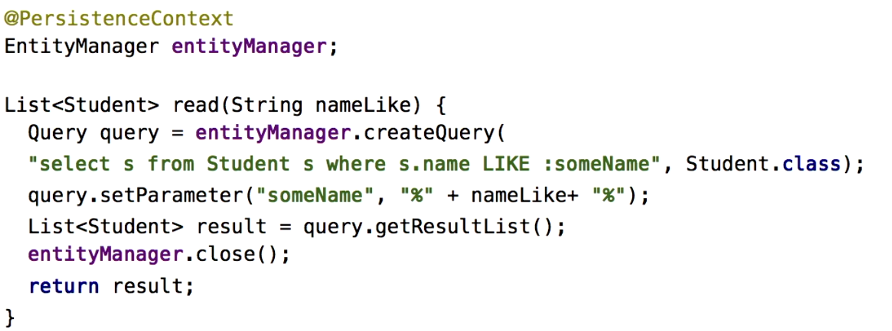
Going from C&C++ to Java, now portable applications can be written once and run anywhere without rebuild.

J2EE, to build enterprise solutions without needing to code the supporting infrastructure. Hibernate eliminated manually mapping logical objects to physical databases.

Asynchronous JavaScript and XML, also known as AJAX, websites became truly interactive like desktop user interfaces.

RESTful web services significantly reduce multi machine communication complexity. Because of better features, less coding Spring Data, and specifically Spring Data JPA, is a game changer. While JPA is great, it still requires repetitive boilerplate code and heavy lifting to make it bug free.

**Example of using JPA:**

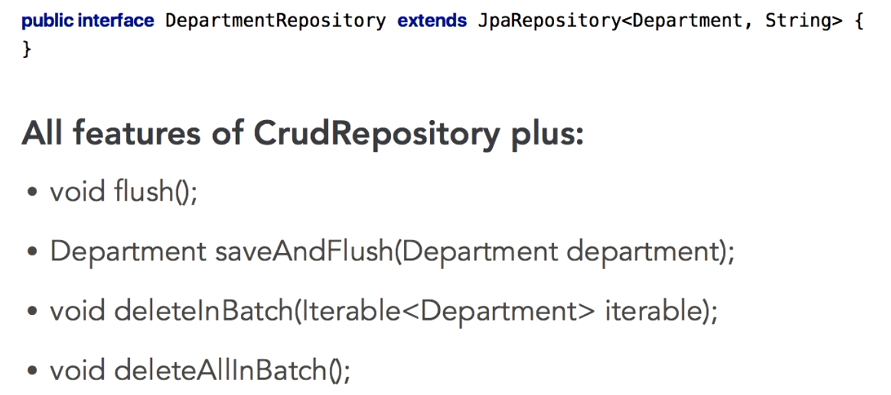


This code is very messy. It's awkward passing in the entity class type to create a query object and every parameter is set one at a time and the query is not run until we invoke get result list or get single result. Worst of all, it's only after running this code that problems are detected.

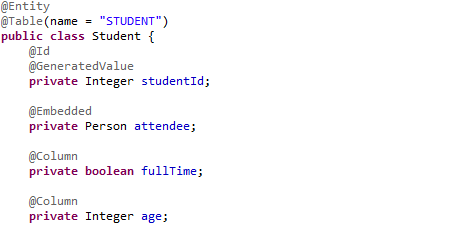
**Spring Data Repository Interfaces:**

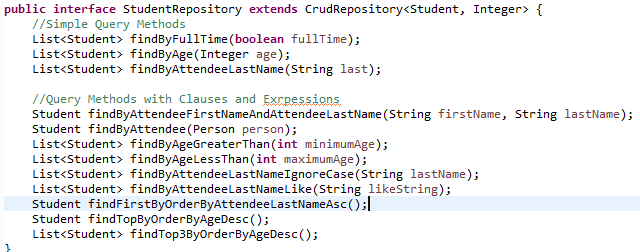
Spring Data abstracts away from any data source by following a repository pattern.

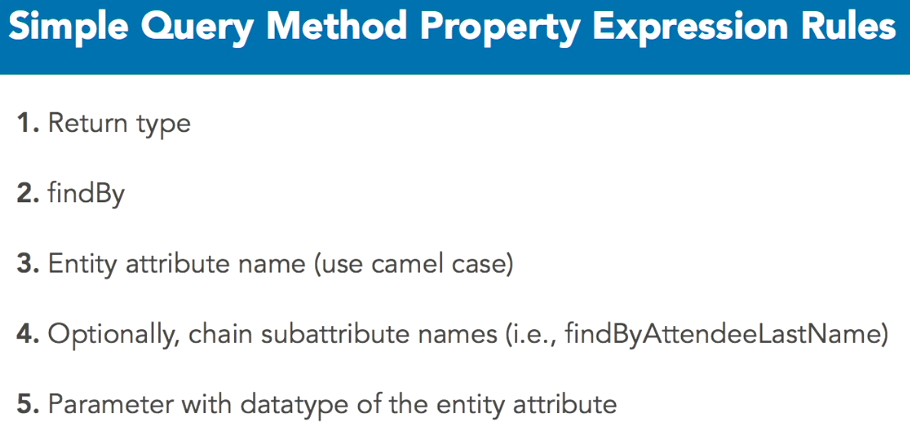
The JpaRepository interface is an example of this. JpaRepository provides the methods of CrudRepository because it extends from CrudRepository but it also provides Jpa specific methods.

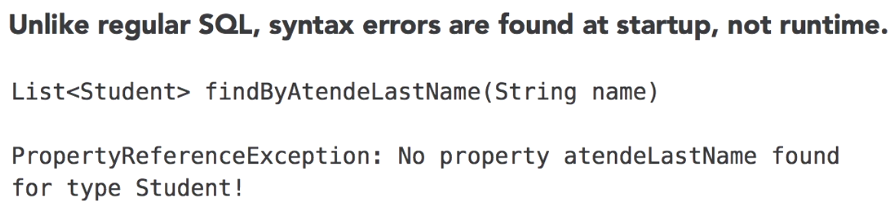


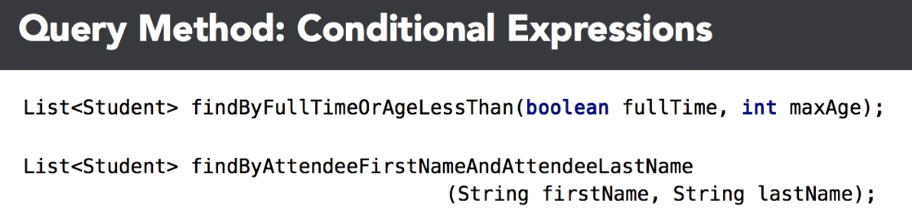
**Example of Student Class:**

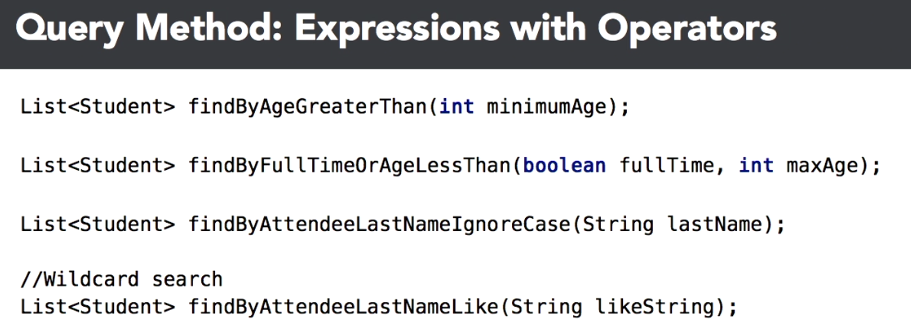






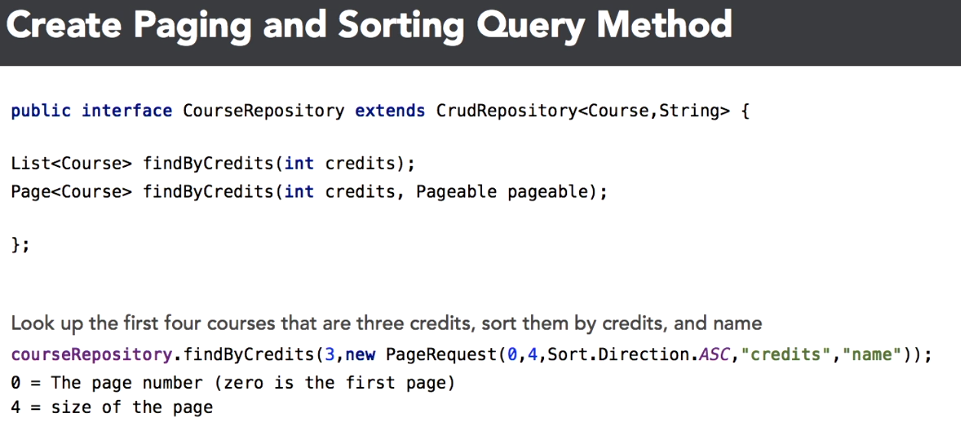


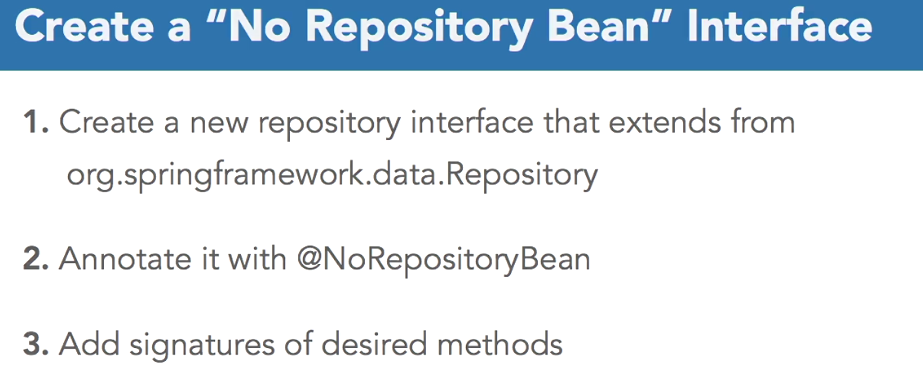




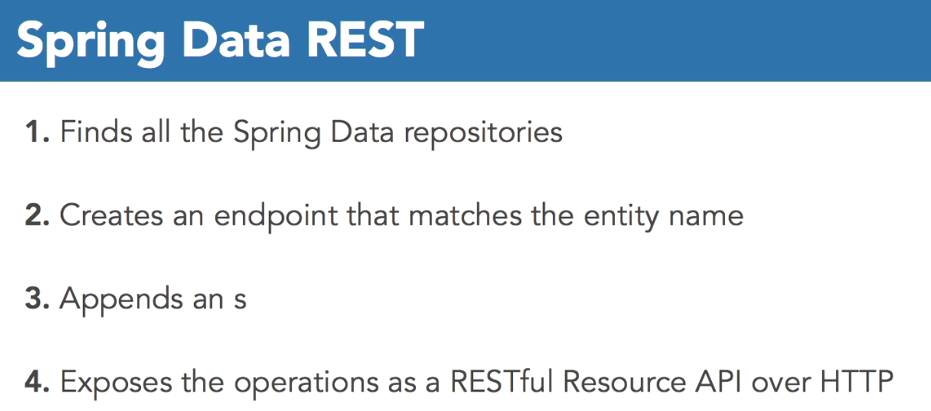
<https://docs.spring.io/spring-data/jpa/docs/2.0.0.M1/reference/html/#repository-query-keywords>







**Spring Data Repository:**



Spring Data rest is a module for exposing Spring Data repositories as hypermedia-driven restful web services. And it's amazing. Like Spring Data commons, which extrapolates the underlying backend data store services from a CrudRepository, Spring Data REST exposes those same repositories as web services with no coding or extra configuration. At application startup, Spring Data REST finds all the available Spring Data repositories, creates an endpoint that matches that entity name, appends an s to the endpoint, and exposes the operations as RESTful Resource APIs over HTTP.

