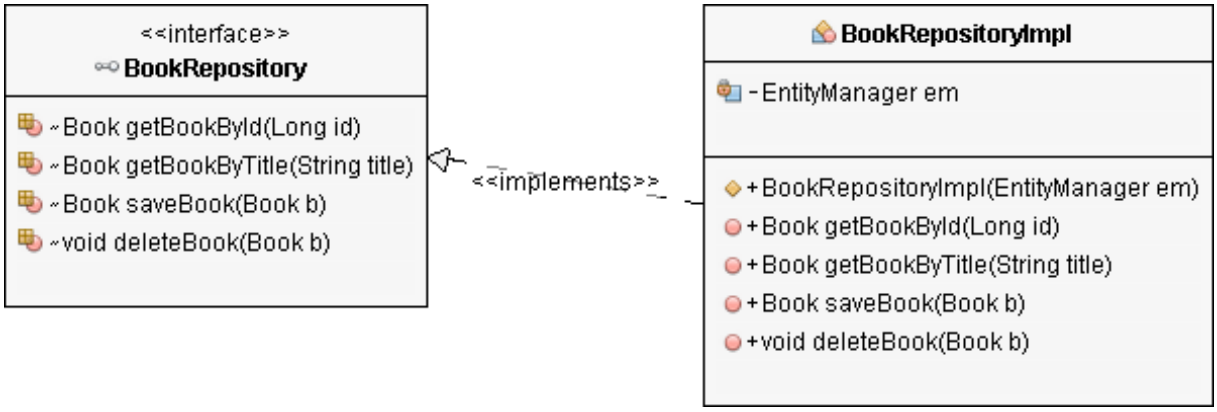
The repository pattern is extremely popular. In its modern interpretation, it abstracts the data store and enables your business logic to define read and write operations on a logical level. It does that by providing a set of methods to read, persist, update and remove an entity from the underlying data store.

Explaining the repository pattern

The repository pattern is pretty simple. An interface defines the repository with all logical read and write operations for a specific entity. You can see an example of such a repository interface in the diagram.

The interface gets implemented by one or more classes that provide data store specific implementations of each interface method.



In my experience, it only rarely happens that you need to support more than one data store. So, you could argue that this pattern creates an overengineered persistence layer. But the interface abstraction also enables frameworks to generate huge parts of the required code.

Implementing the repository pattern

In most enterprise projects, you only need to define the repository interfaces. Spring Data JPA and Apache DeltaSpike Data can generate standard repository implementations for you. You just need to provide your own implementation, if your implementation gets

especially complex. I will show you more of that in the following articles of this series.

But for now, let's implement the repository pattern without any frameworks. That makes the pattern easier to understand and highlights the benefits of frameworks that generate repetitive parts of the implementation.

Defining the repository interface

Let's implement the same *BookRepository* interface as I showed you in the diagram. It defines 4 methods that you can use to:

- save a new or changed entity (Please keep in mind that Hibernate detects and persists all changes of managed entities automatically. So, you don't need to call the save method after you changed any entity attributes),
- delete an entity,
- find an entity by its primary key and
- find an entity by its title.

```
public interface BookRepository {

        Book getBookById(Long id);

        Book getBookByTitle(String title);

        Book saveBook(Book b);

        void deleteBook(Book b);
}
```

Implementing the repository with JPA and Hibernate

In the next step, you can implement the *BookRepository* interface.

```
public class BookRepositoryImpl implements BookRepository {

	private EntityManager em;

	public BookRepositoryImpl(EntityManager em) {
		this.em = em;
	}

	@Override
	public Book getBookById(Long id) {
		return em.find(Book.class, id);
	}

	@Override
	public Book getBookByTitle(String title) {
		TypedQuery<Book> q = em.createQuery("SELECT b FROM Book b
WHERE b.title = :title", Book.class);
		q.setParameter("title", title);
		return q.getSingleResult();
	}

	@Override
	public Book saveBook(Book b) {
		if (b.getId() == null) {
			em.persist(b);
		} else {
			b = em.merge(b);
		}
		return b;
	}
```

```
        @Override
        public void deleteBook(Book b) {
                if (em.contains(b)) {
                        em.remove(b);
                } else {
                        em.merge(b);
                }
        }
}
```

If you ever called a JPQL query or persisted an entity in your business layer, the code of my repository implementation should look familiar. There is no big difference between implementing these operations in your business code or as part of a repository implementation.

In this example, the only noticeable difference is the implementation of the saveBook(Book b) method. You can call this method to persist a new entity or to merge an existing one. So, you need to detect if the method got called with a new or an existing entity. In this example, I let Hibernate generate the primary key values. So, the id attribute of all new entities should be null. If it isn't null, it should be an existing entity which then gets merged into the persistence context.