DZone Refcardz

CONTENTS INCLUDE:

Core Spring Data

By: Oliver Gierke

ABOUT THE SPRING DATA PROJECT

The Spring Data project is part of the ecosystem surrounding the Spring Framework and constitutes an umbrella project for advanced data access related topics. It contains modules to support traditional relational data stores (based on plain JDBC or JPA), NoSQL ones (like MongoDB, Neo4j or Redis), and big data technologies like Apache Hadoop. The core mission of the project is to provide a familiar and consistent Spring-based programming model for various data access technologies while retaining store-specific features and capabilities.

General Themes

Infrastructure Configuration Support

A core theme of all the Spring Data projects is support for configuring resources to access the underlying technology. This support is implemented using XML namespaces and support classes for Spring JavaConfig allowing you to easily set up access to a Mongo database, an embedded Neo4j instance, and the like. Also, integration with core Spring functionality like JMX is provided, meaning that some stores will expose statistics through their native API, which will be exposed to JMX via Spring Data.

Object Mapping Framework

Most of the NoSQL Java APIs do not provide support to map domain objects onto the stores' data model (e.g., documents in MongoDB, or nodes and relationships for Neo4j). So, when working with the native Java drivers, you would usually have to write a significant amount of code to map data onto the domain objects of your application when reading, and vice versa on writing. Thus, a core part of the Spring Data project is a mapping and conversion API that allows obtaining metadata about domain classes to be persisted and enables the conversion of arbitrary domain objects into store-specific data types.

Template APIs

On top of the object mapping API, we'll find opinionated APIs in the form of template pattern implementations already well-known from Spring's JdbcTemplate, JmsTemplate, etc. Thus, there is a RedisTemplate, a MongoTemplate, and so on. These templates offer helper methods that allow you to execute commonly needed operations like persisting an object with a single statement while automatically taking care of appropriate resource management and exception translation. Beyond that, they expose callback APIs that allow you to access the store-native APIs while still getting exceptions translated and resources managed properly.

Repository Abstraction

These features already provide us with a toolbox to implement a data access layer like we're used to with traditional databases. The upcoming sections will guide you through this functionality. To simplify the development of a data access layer, Spring Data provides a repository abstraction on top of the template implementation. This reduces the effort to implement data access objects to writing a plain interface definition for the most common scenarios like performing standard CRUD operations as well as executing. This data access layer abstraction provides a layer of portability for CRUD operations across the different stores but doesn't limit you to gain access to store specific features like geo-spatial queries in MongoDB.

CONFIGURATION SUPPORT

At the very lowest level, Spring Data provides support to easily configure infrastructure components and enable Spring Data features like the repository support. The individual modules provide this support through a Spring XML namespace and—where reasonable—the JavaConfig equivalents implemented as @Enable... annotations.

JPA

XML element	Description
<jpa:repositories />	Enables Spring Data repositories support for repository interfaces underneath the package configured in the base-package attribute. JavaConfig equivalent is @EnableJpaRepositories.
<jpa:auditing />	Enables transparent auditing of JPA managed entities. Note that this requires the AuditingEntityListener applied to the entity (either globally through a declaration in orm.xml or through @EntityListener on the entity class).

MongoDB

For Spring Data MongoDB XML namespace elements not mentioning a dedicated @Enable annotation alternative, you usually declare an @Bean-annotated method and use the plain Java APIs of the classes that would have otherwise been set up by the XML element. Alternatively, you can use the JavaConfig base class AbstractMongoConfiguration that Spring Data MongoDB ships for convenience.

XML element	Description
<mongo:db-factory />	One stop shop to set up a Mongo instance pointing to a particular database instance. For advanced-use cases define a <mongo:mongo /> extenally and refer to it using a mongo-ref attribute.
<mongo:mongo />	Configures a Mongo instance. Supports basic attributes like host, port, write concern etc. Configure more advanced options through the nested <mongo:options /> element. In JavaConfig simply declare an @Bean method returning a Mongo instance.
<mongo:mapping-converter />	Configures a MappingMongoConverter. Allows enabling scanning for entity types on bootstrap (through base-package attribute) and registering custom converters through nested <mongo:custom-converters /> element.

XML element	Description
<mongo:repositories />	Enables Spring Data MongoDB repositories support for repository interfaces underneath the package configured in the base-package attribute. The JavaConfig equivalent is @EnableMongoRepositories.
<mongo:auditing />	Enables transparent auditing of MongoDB persisted domain objects.
<mongo:jmx />	Enables exposing MongoDB statistics and configuration as JMX MBeans.

Neo4j

For Spring Data Neo4j XML namespace elements not mentioning a dedicated @Enable annotation alternative, you usually declare an @Bean-annotated method and use the plain Java APIs of the classes that would have been set up by the XML element otherwise. Alternatively, you can use the JavaConfig base class Neo4jConfiguration that Spring Data Neo4j ships for convenience.

XML element	Description
<neo4j:config />	Configures connection to the Neo4j data store and entity types to be scanned for on bootstrap.
<neo4j:repositories />	Enables Spring Data Neo4j repositories support for repository interfaces underneath the package configured in the base-package attribute. The JavaConfig equivalent is @EnableNeo4jRepositories.
<neo4j:auditing />	Enables transparent auditing of Neo4j mapped entities.

OBJECT MAPPING

If you have been working with JPA you are already familiar with annotation based mapping of classes onto relational database tables. When working with NoSQL stores you need similar means to configure how your classes are mapped onto MongoDB documents or Neo4j nodes and relationships.

JPA

Here's what a simple JPA-mapped Customer class could look like:

```
@Entity
public class Customer {

  @Id @GeneratedValue
  private Long id;
  private String firstname, lastname;

  @Column(unique = true)
  private EmailAddress emailAddress;

  @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
  @JoinColumn(name = "customer_id")
  private Set<Address> addresses = new HashSet<Address>();

  …
}
```

As the JPA persistence provider is taking care of the object-relational mapping, Spring Data JPA supports all JPA mapping annotations and essentially does not deal with the mapping at all.

MongoDB

To persist the same Customer type into MongoDB you would use the Spring Data mapping annotations as shown below:

```
@Document
public class Customer {

  @Id
  private BigInteger id;
  private String firstname, lastname;

  @Field("email")
  @Indexed(unique = true)
  private EmailAddress emailAddress;
  private Set<Address> addresses = new HashSet<Address>();

  …
}
```

Spring Data MongoDB Mapping Annotations

Annotation	Description
@Id	(optional) Determines the identifier property of a class. If not explicitly used, properties named id or _id will be considered IDs.
@Document	(optional) Marks a class as to be stored as document. Will only be used to component scan for document classes to build mapping metadata on application context startup.
@DBRef	Defines an object to be stored in a separate collection instead of being embedded.
@Transient	Excludes a property from being persisted.
@Indexed	Defines an index to be created for the property annotated.
@CompoundIndex	Defines a compound index to be created.
@GeoSpatialIndexed	Defines a geo-spatial index to be created for the given property. Property type needs to be double[] or Point.
@PersistenceConstructor	(optional) Selects one of multiple constructors to be used for object instantiation on read operations. For disambiguation purposes only.
@Value	(optional) Allows customization of the value to be used for a constructor parameter during object construction. Refer to the just read DBObject through #root.
@Field	(optional) Allows customization of the field name to be used and the field ordering inside the resulting document.

Neo4j

The same Customer entity could also be persisted to a Neo4j data store using Spring Data mapping annotations is shown below:

```
@NodeEntity
public class Customer {

  @GraphId
  private Long id;
  private String firstName, lastName;

  @Indexed(unique = true)
  private String emailAddress;

  @RelatedTo(type = "ADDRESS")
  private Set<Address> addresses = new HashSet<Address>();

  …
}
```

Spring Data Neo4j Mapping Annotations

Annotation	Description
@GraphId	Determines the identifier property of a class.
@Transient	Excludes a property from being persisted.
@NodeEntity	Declares an entity class to be backed by a node.

Annotation	Description
@RelationshipEntity	Declares an entity class to be backed by a relationship.
@StartNode / @ EndNode	To be used in an entity declared as @ RelationshipEntity. Defines the property to capture the start or end node of a relationship.
@RelatedTo / @ RelatedToVia	Annotation for entity fields that relate to other entities via relationships.
@Indexed	Configures indexing to be applied for the annotated property.

TEMPLATE APIS

Developers are likely familiar with the various template pattern implementations available in the core Spring framework, like JdbcTemplate and JmsTemplate. The Spring Data projects extend the usage of this pattern for stores that do not provide a dedicated object mapping data access API already, like MongoDB and Neo4j. The Spring Data template classes have 3 major responsibilities:

1. Transparent integration of the object-to-store mapping.
2. Resource management. Reliably acquiring and releasing connections, in the case of exceptions being thrown.
3. Exception translation. The templates automatically convert the persistence technology specific exceptions being thrown into Spring's DataAccessException hierarchy to prevent the persistence technology from leaking into client code.

The template implementations usually expose three different groups of methods:

1. Standard use-case methods like findOne(…) and findAll(…) taking, for example, query and the type of object to be retrieved.
2. Store specific functionality like createCollection(…) or geoNear(…) for MongoDB, or createNode(…) or traverse(…) in the case of Neo4j.
3. Low-level methods (usually called execute(…)) accepting callback interfaces to give you access to the store's native API but still taking care of resource management and exception translation.

These implementations provide a solid foundation to consistently implement data access layers for different stores and exposing the individual stores features.

REPOSITORIES

Even with a great API to build upon, implementing repositories involves a lot of repetitive boilerplate code. The Spring Data modules ship with an interface based programming model to help developers reduce the amount of code written to the minimum possible.

Quickstart

Working with the repository abstraction generally follows the following pattern:

1. Create interface annotated with @RepositoryDefinition or extending one of Spring Data's base repository interfaces
2. Add query methods and customize as needed
3. Potentially add custom implementation if necessary

To create a repository for one of the Customer domain classes listed above you'd simply declare an interface extending e.g. the Repository marker interface and add basic query methods to it.

```
interface CustomerRepository extends Repository<Customer, Long> {

  Customer findByEmailAddress(EmailAddress email);

  List<Customer> findByAddressCity(String city);

  Page<Customer> findByLastnameLike(String lastname, Pageable
pageable);
}
```

The Query Derivation Mechanism

A sole declaration of a query method will cause the Spring Data repository infrastructure to parse the method name into a store-specific query. The method names just shown consist of a prefix (terminated with the word by), a property reference (direct, as in EmailAddress, or referring to nested properties as in AddressCity) possibly augmented by a keyword that defines how the parameters handed into the method are bound to the property reference similar to frameworks like Grails.

The following table lists all keywords available for all the stores that generally support the repository abstraction:

Logical keyword	Keyword expressions
AFTER	After, IsAfter (date types only)
BEFORE	Before, IsBefore (date types only)
BETWEEN	Between, IsBetween (date and numeric types only)
CONTAINING	Containing, IsContaining, Contains (String or collection types only)
ENDING_WITH	EndingWith, IsEndingWith, EndsWith (String types only)
EXISTS	Exists
FALSE	False (boolean properties only)
GREATER_THAN	GreaterThan, IsGreaterThan (comparable types only)
GREATER_THAN_EQUALS	GreaterThanEqual, IsGreaterThanEqual (compüarable types only)
IN	In, IsIn (expects collection or array parameter)
IS	Is, Equals, (or no keyword at all)
IS_NOT_NULL	NotNull, IsNotNull
IS_NULL	Null, IsNull
LESS_THAN	LessThan, IsLessThan (comparable types)
LESS_THAN_GREATER	LessThanGreater, IsLessThanGreater (comparable types only)
LIKE	Like, IsLike (expects the store-specific like syntax, prefer CONTAINING, STARTS_WITH, ENDS_WITH)
NOT	Not, IsNot (boolean types only)
NOT_IN	NotIn, IsNotIn (expects collection or array parameter)
NOT_LIKE	NotLike, IsNotLike
STARTING_WITH	StartingWith, IsStartingWith, StartsWith (String types only)
TRUE	True, IsTrue (boolean types only)

The query derivation mechanism will create a query per method on application bootstrap time, and will make sure it's syntactically correct. Depending on which store you bootstrap, the repository infrastructure for the resulting query for findByAddressCity(…) would look as follows:

Store	Keyword expressions
JPA	Select c from Customer c where c.address.city = ?1
MongoDB	{ “address.city“ : ?0 }
Neo4j	start address=node:Address(city={0}) match address<-[:LIVES_AT]-customer return customer

MongoDB-Specific Keywords

As MongoDB has support for geospatial functionality, we also expose specific keywords to work with it transparently. Note that this requires to the referenced properties to be annotated with @GeoSpatialIndexed.

Logical keyword	Keyword expressions
NEAR	Near, IsNear (expects parameter of type Point or double[])
WITHIN	Within, IsWithin (expects parameter of type Box, Circle and an optional Distance)
REGEX	Regex, MatchesRegex, Matches (String types only)

Customizing Query Execution

In case the query derivation mechanism falls short of your requirements, the individual modules ship with an @Query annotation that allows you to customize the query to be executed.

```
interface CustomerRepository extends Repository<Customer, Long> {

  @Query("select c from Customer c where c.emailAddress = ?1")
  Customer findByEmailAddress(EmailAddress email);
}
```

If you want to keep the query out of your repository interface you can also simply use JPA named queries and follow the naming convention of $domainClass.$methodName, so Customer.findByEmailAddress in this case. Beyond that, Spring Data supports a named query mechanism across all store types, by declaring queries in a $store-named-queries.properties file in META-INF, e.g. mongodb-named-queries.properties. The naming conventions for the queries themselves follow the pattern described above.

Choosing A Repository Base Interface

In the sample just shown we extend the Repository interface provided by Spring Data. It is essentially a marker interface that captures the domain and ID type managed. There are a few other base repositories you might want to extend from to pull in additional functionality into your repository:

Base interface	Description
Repository	Marker interface to communicate the domain and ID types to the infrastructure.
CrudRepository	Exposes CRUD (Create, Read, Update, Delete) methods for the configured domain type. Extends Repository.
PagingAndSortingRepository	Extends CrudRepository and exposes additional methods to pass sorting information to the repository and lookup entities in a paginated way.
JpaRepository, MongoDbRepository, Neo4jRepository	Usually extend from PagingAndSortingRepository and provide additional, store-specific methods.

We generally recommend not to extend the store-specific interfaces as you reveal the persistence technology used to the client and couple your code to it.

Pagination and Sorting

The PagingAndSortingRepository provides an API that captures information about sorting as well as the definition and results of page-by-page access. The core abstractions of the API are Sort and Pageable that define how to page as well as a Page object to capture results. The Page class provides additional metadata about the page of data—often shown in a UI—such as how many pages are there in total, or whether it is the first or last page. With our CustomerRepository implemented PagingAndSortingRepository, we could write the following code:

```
// Create request for the 2nd page by a page size of 10
Sort sort = new Sort(Direction.ASC, "lastname", "firstname");
Pageable pageable = new PageRequest(1, 10, sort);
Page<Customer> customers = repository.findAll(pageable);

assertThat(customers.isFirstPage(), is(false));
assertThat(customers.hasNextPage(), is(true));

for (Customer customer : customers) {
  // … do something with the Customer
}
```

The repository abstraction considers method parameters of type Pageable and Sort as special parameters that need to augment either the query (sorting) or the query execution (pagination). Here's the query method using the pagination from the sample repository already shown above:

```
interface CustomerRepository extends Repository<Customer, Long> {

  // … other methods omitted
  Page<Customer> findByLastnameLike(String lastname, Pageable
pageable);
}
```

This definition would make sure the Page instance is populated with the necessary metadata (potentially by issuing additional queries) and the limiting of the result set is added to the query by using store specific API underneath. We'll see how we can easily obtain Pageable instances from web request parameters later on.

ADVANCED FEATURES

Spring Data also provides advanced integration with third-party projects and technologies.

Querydsl

The Querydsl project (http://querydsl.org) aims to bring LINQ-style (language integrated queries) capabilities to the Java platform. It is a bit like the JPA criteria API but not tied to JPA, a lot less verbose and thus much more comfortable to use.

Querydsl is centered around a meta-model generated from domain classes that can be used to define predicates over the entities. These predicates can then be executed via a Spring Data Repository.

Metamodel Generation

The meta-model is generated through a step in the build process. Querydsl currently provides integration for Maven and Ant. Here's the sample configuration for JPA and Maven:

```
<plugin>
  <groupId>com.mysema.maven</groupId>
  <artifactId>apt-maven-plugin</artifactId>
  <version>1.0.9</version>
  <executions>
    <execution>
      <goals>
        <goal>process</goal>
      </goals>
      <configuration>
        <outputDirectory>
          target/generated-sources/java
        </outputDirectory>
        <processor>
          com.mysema.query.apt.jpa.JPAAnnotationProcessor
        </processor>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Predicate Definition

This configuration will inspect the JPA mapping annotations and create meta-model classes that can now be used as follows:

```
QCustomer customer = Qcustomer.customer;
Predicate predicate = customer.lastname.endswith(…).
 or(customer.firstname.startsWith(…));
```

You see we can easily define constraints on the entities using a fluent API.

Predicate Execution

To eventually execute the predicates just defined, Spring Data provides a QueryDslPredicateExecutor interface that your repository can extend to expose the necessary API to your clients:

```
interface CustomerRepository extends Repository<Customer, Long>,
  QueryDslPredicateExecutor<Customer, Long> {

}
```

A client would then look as follows:

```
class RepositoryClient {

  @Autowired CustomerRepository repository;

  public void buisnessMethod(String firstname, String lastname) {

    QCustomer customer = Qcustomer.customer;

    List<Customer> customers = repository.findAll(
      customer.lastname.endswith(lastname).or(
      customer.firstname.startsWith(firstname));

    …
  }
}
```

CDI Integration

The Spring Data modules discussed here all ship with a CDI extension that allows using the repository abstraction in a JavaEE environment that uses CDI for dependency injection.

1. Have the necessary Spring Data JARs in your classpath.
2. Declare necessary infrastructure components as CDI beans using the @Produces/@Disposes annotation.
3. Declare repository interfaces as described above.
4. Inject the repository instances into your CDI managed beans using @Inject.

JPA

In JPA the core infrastructure abstraction is the EntityManager. Working in a standalone CDI environment the EntityManager is created by an EntityManagerFactory. It can be exposed to the infrastructure as follows:

```
class EntityManagerFactoryProducer {

  @Produces
  @ApplicationScoped
  public EntityManagerFactory createEntityManagerFactory() {
    return Persistence.createEntityManagerFactory("my-pu");
  }

  public void close(
    @Disposes EntityManagerFactory entityManagerFactory) {
    entityManagerFactory.close();
  }
}
```

Once that is done, you can get a hold of your repository instance by injecting it into your CDI bean:

```
class RepositoryClient {

  @Inject
  CustomerRepository repository;

  public void businessMethod(EmailAddress email) {
    Customer customer = repository.findByEmailAddress(email);
    …
  }
}
```

MongoDB

In the case of MongoDB, essentially all you need to change is the infrastructure setup. Instead of configuring an EntityManagerFactory you expose a MongoTemplate instance to CDI:

```
class MongoTemplateProducer {

  @Produces
  @ApplicationScoped
  public MongoOperations createMongoTemplate() throws
    UnknownHostException, MongoException {

    MongoDbFactory factory =
      new SimpleMongoDbFactory(new Mongo(), "database");
    return new MongoTemplate(factory);
  }
}
```

The injection of the repository into the client looks identical to the code shown for JPA:

```
class RepositoryClient {

  @Inject
  CustomerRepository repository;

  public void businessMethod(EmailAddress email) {
    Customer customer = repository.findByEmailAddress(email);
    …
  }
}
```

Integration with Spring and SpringMVC

The Spring Data Commons module ships with a variety of support classes that ease the general interaction with the core Spring framework and Spring MVC in particular. Here are the interesting types:

Base interface	Description
DomainClassConverter	Allows conversion of a String ID into the entity with the given ID using the findOne(") method on the repository registered for the domain type.
DomainClassPropertyEditor	Legacy version of DomainClassConverter.
SortHandlerMethodArgument Resolver	Automatically creates a Sort instance from HttpServletRequest parameters.
PageableHandlerMethod ArgumentResolver	Automatically creates a Pageable instance from HttpServletRequest parameters.

Converter / PropertyEditor Support

SpringMVC controller methods are extremely flexible and allow you to refer to URI path segments, request parameters etc. in a type-safe way. With the DomainClassConverter registered in the application context, you can write a controller method like this:

```
@Controller
public class CustomerController {

  @RequestMapping("/customers/{id}")
  public String showUserForm(
    @PathVariable("id") Customer customer, Model model) {

    …
  }
}
```

The interesting part here is that for a call to /users/47, you immediately get the User instance with ID 47 handed into the method in case it is managed by a Spring Data repository exposing a findOne(…) method. You can simply register the converter through the appropriate callback method in your SpringMVC configuration:

```
class WebConfiguration extends WebMvcConfigurationSupport {

  @Bean
  public DomainClassConverter<?> domainClassConverter() {
    return new DomainClassConverter<FormattingConversionService>(
      mvcConversionService());
  }
}
```

Pagination and Sorting

The same approach can be used to transparently create instances for Sort and Pageable from request parameters. This doesn't even require the usage of the repository abstraction. All you need to do is configure the appropriate HandlerMethodArgumentResolver implementations in your SpringMVC configuration:

```
class WebConfiguration extends WebMvcConfigurationSupport {

  protected void addArgumentResolvers(
    List<HandlerMethodArgumentResolver> resolvers) {

    resolvers.add(new SortHandlerMethodArgumentResolver());
    resolvers.add(new PageableHandlerMethodArgumentResolver());
  }
}
```

This allows SpringMVC controller methods that look like this:

```
@Controller
public class UserController {

  @Autowired CustomerRepository repository;

  @RequestMapping("/users")
  public String showUserForm(Pageable pageable, Model model) {

    Page<Customer> customers = repository.findAll(pageable);
    …
  }
}
```

The resolver implementations can be customized as you see fit but by default they will lookup the following request parameters to build up the Pageable and Sort instances:

Parameter	Description
page (0‥1)	The number of the page to be retrieved. (0-indexed, defaults to 0)
size (0‥1)	The size of the page to be retrieved. (defaults to 20)
sort (0‥n)	A sort expression with comma-separated field references, terminated by asc or desc to define the order (optional). E.g /?sort=firstname,asc&lastname=desc

To be able to selectively customize the defaults for Pageable and Sort instances, the controller method arguments can be annotated with the following annotations:

Annotation	Description
@Qualifier	Allows definition of a qualifier in case multiple pages or sorts need to be handed into a single method. Will get prepended to the parameters (foo_page, foo_size etc.).
@PageableDefault	Customizes the defaults to be used for the Pageable instance in case no request parameters are present. If not set, global defaults apply.
@SortDefault	Customizes the defaults to be used for the Sort instance in case no request parameters are present. If not set, the Sort will be null.

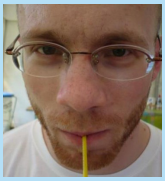
RESOURCES

Spring Data REST

The Spring Data REST project provides support to automatically export repository-managed entities in a hypermedia-driven way. By default it renders customizable JSON representations for your domain classes but also provides a lot of extension and customization hooks to enable validation, security and integration of custom code. For more details, see: https://github.com/SpringSource/spring-data-rest

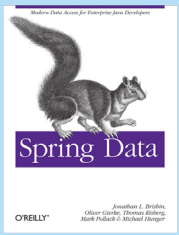
Project home: http://www.springsource.org/spring-data

ABOUT THE AUTHOR



Oliver Gierke is an engineer at SpringSource, member of the JPA 2.1 expert group, and project lead of the Spring Data JPA, MongoDB and core modules. He has been developing enterprise applications and open source projects for over 6 years. Oliver's work centers around architecture, Spring, and persistence technologies. He is regularly speaks at German and international conferences and has written numerous technical papers on software development.

RECOMMENDED BOOK



Through several sample projects, you'll learn how Spring Data provides a consistent programming model that retains NoSQL-specific features and capabilities, and helps you develop Hadoop applications across a wide range of use-cases such as data analysis, event stream processing, and workflow.

BUY HERE

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1
50795
9 781936 502806
$7.95

Version 1.0