Caching Data with Spring

This guide walks you through the process of enabling caching on a Spring managed bean.

What you'll build

You'll build an application that enables caching on a simple book repository.

What you'll need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 2.3+ or Maven 3.0+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step, or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Build with Gradle.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-caching.git`
- cd into `gs-caching/initial`
- Jump ahead to Create a book repository.

When you're finished, you can check your results against the code in `gs-caching/complete`.

Create a book repository

First, let's create a very simple model for your book

`src/main/java/hello/Book.java`

```
package hello;

public class Book {

    private String isbn;
    private String title;

    public Book(String isbn, String title) {
```

```
        this.isbn = isbn;
        this.title = title;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public String toString() {
        return "Book{" + "isbn='" + isbn + '\'' + ", title='" + title + '\'' + '}';
    }

}
```

And a repository for that model:

src/main/java/hello/BookRepository.java

```
package hello;

public interface BookRepository {

    Book getByIsbn(String isbn);

}
```

You could have used Spring Data to provide an implementation of your repository over a wide range of SQL or NoSQL stores, but for the purpose of this guide, you will simply use a naive implementation that simulates some latency (network service, slow delay, etc).

src/main/java/hello/SimpleBookRepository.java

```
package hello;

import org.springframework.stereotype.Component;

@Component
public class SimpleBookRepository implements BookRepository {
```

```
    @Override
    public Book getByIsbn(String isbn) {
        simulateSlowService();
        return new Book(isbn, "Some book");
    }

    // Don't do this at home
    private void simulateSlowService() {
        try {
            long time = 3000L;
            Thread.sleep(time);
        } catch (InterruptedException e) {
            throw new IllegalStateException(e);
        }
    }

}
```

`simulateSlowService` is deliberately inserting a three second delay into each `getByIsbn` call. This is an example that later on, you'll speed up with caching.

Using the repository

Next, wire up the repository and use it to access some books.

`src/main/java/hello/Application.java`

```
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration` tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration` tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- Normally you would add `@EnableWebMvc` for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This flags the application as a web application and activates key behaviors such as setting up a `DispatcherServlet`.
- `@ComponentScan` tells Spring to look for other components, configurations, and services in the `hello` package, allowing it to find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there wasn't a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn't have to deal with configuring any plumbing or infrastructure.

There is also a `CommandLineRunner` that injects the `BookRepository` and calls it several times with different arguments.

`src/main/java/hello/AppRunner.java`

```
package hello;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class AppRunner implements CommandLineRunner {

    private static final Logger logger = LoggerFactory.getLogger(AppRunner.class);

    private final BookRepository bookRepository;

    public AppRunner(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @Override
    public void run(String... args) throws Exception {
        logger.info(".... Fetching books");
        logger.info("isbn-1234 -->" + bookRepository.getByIsbn("isbn-1234"));
        logger.info("isbn-4567 -->" + bookRepository.getByIsbn("isbn-4567"));
        logger.info("isbn-1234 -->" + bookRepository.getByIsbn("isbn-1234"));
        logger.info("isbn-4567 -->" + bookRepository.getByIsbn("isbn-4567"));
        logger.info("isbn-1234 -->" + bookRepository.getByIsbn("isbn-1234"));
        logger.info("isbn-1234 -->" + bookRepository.getByIsbn("isbn-1234"));
    }

}
```

If you try to run the application at this point, you'll notice it's quite slow even though you are retrieving the exact same book several times.

```
2014-06-05 12:15:35.783  ... : .... Fetching books
2014-06-05 12:15:40.783  ... : isbn-1234 -->Book{isbn='isbn-1234', title='Some book'}
2014-06-05 12:15:43.784  ... : isbn-1234 -->Book{isbn='isbn-1234', title='Some book'}
2014-06-05 12:15:46.786  ... : isbn-1234 -->Book{isbn='isbn-1234', title='Some book'}
```

As can be seen by the timestamps, each book took about three seconds to retrieve, even though it's the same title being repeatedly fetched.

Enable caching

Let's enable caching on your `SimpleBookRepository` so that the books are cached within the `books` cache.

`src/main/java/hello/SimpleBookRepository.java`

```
package hello;

import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class SimpleBookRepository implements BookRepository {

    @Override
    @Cacheable("books")
    public Book getByIsbn(String isbn) {
        simulateSlowService();
        return new Book(isbn, "Some book");
    }

    // Don't do this at home
    private void simulateSlowService() {
        try {
            long time = 3000L;
            Thread.sleep(time);
        } catch (InterruptedException e) {
            throw new IllegalStateException(e);
        }
    }

}
```

You now need to enable the processing of the caching annotations

`src/main/java/hello/Application.java`

```
package hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@SpringBootApplication
@EnableCaching
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

The `@EnableCaching` annotation triggers a post processor that inspects every Spring bean for the presence of caching annotations on public methods. If such an annotation is found, a proxy is automatically created to

intercept the method call and handle the caching behavior accordingly.

The annotations that are managed by this post processor are `Cacheable`, `CachePut` and `CacheEvict`. You can refer to the javadocs and the documentation for more details.

Spring Boot automatically configures a suitable `CacheManager` to serve as a provider for the relevant cache. See the Spring Boot documentation for more details.

Our sample does not use a specific caching library so our cache store is the simple fallback that uses `ConcurrentHashMap`. The caching abstraction supports a wide range of cache library and is fully compliant with JSR-107 (JCache).

Build an executable JAR

You can run the application from the command line with Gradle or Maven. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you are using Gradle, you can run the application using `./gradlew bootRun`. Or you can build the JAR file using `./gradlew build`. Then you can run the JAR file:

```
java -jar build/libs/gs-caching-0.1.0.jar
```

If you are using Maven, you can run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

```
java -jar target/gs-caching-0.1.0.jar
```

> The procedure above will create a runnable JAR. You can also opt to build a classic WAR file instead.

Test the application

Now that caching is enabled, you can execute it again and see the difference by adding additional calls with or without the same isbn. It should make a huge difference.

```
2016-09-01 11:12:47.033  .. : .... Fetching books
2016-09-01 11:12:50.039  .. : isbn-1234 -->Book{isbn='isbn-1234', title='Some book'}
2016-09-01 11:12:53.044  .. : isbn-4567 -->Book{isbn='isbn-4567', title='Some book'}
2016-09-01 11:12:53.045  .. : isbn-1234 -->Book{isbn='isbn-1234', title='Some book'}
2016-09-01 11:12:53.045  .. : isbn-4567 -->Book{isbn='isbn-4567', title='Some book'}
2016-09-01 11:12:53.045  .. : isbn-1234 -->Book{isbn='isbn-1234', title='Some book'}
2016-09-01 11:12:53.045  .. : isbn-1234 -->Book{isbn='isbn-1234', title='Some book'}
```

This excerpt from the console shows that the first time to fetch each title took three seconds, but each subsequent call was near instantaneous.

Summary

Congratulations! You've just enabled caching on a Spring managed bean.

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.