Chapter 6. Really Digging into Data

Data can be a complex topic, with so much to consider: its structure and relationships with other data; handling, storage, and retrieval options; various applicable standards; database providers and mechanisms; and more. Data may be the most complex aspect of development to which devs are exposed so early in their careers and when learning a new toolchain.

The reason this is often the case is that without data in some form, nearly all applications are meaningless. Very few apps provide any value at all without storing, retrieving, or correlating data.

As something that forms the underpinning for nearly all application value, *data* has attracted a great deal of innovation from database providers and platform vendors. But in many cases, complexity remains: it is a topic with great depth and breadth, after all.

Enter Spring Data. Spring Data's stated mission is "to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store." Regardless of database engine or platform, Spring Data's goal is to make the developer's access to data as simple and as powerful as humanly possible.

This chapter demonstrates how to define data storage and retrieval using various industry standards and leading database engines and the Spring Data projects and facilities that enable their use in the most streamlined and powerful ways possible: via Spring Boot.

Defining Entities

In nearly every case when dealing with data, some form of domain entity is involved. Whether it's an invoice, an automobile, or something else entirely, data is rarely dealt with as a collection of unrelated properties. Inevitably, what we consider useful data are cohesive pools of elements that together constitute a meaningful whole. An automobile—in data or in real life—is really only a useful concept if it's a unique, fully attributed thing.

Spring Data provides several different mechanisms and data access options for Spring Boot applications to use, at a variety of abstraction levels. Regardless of which level of abstraction a developer settles on for any given use case, the first step is defining any domain classes that will be used to handle applicable data.

While a full exploration of Domain-Driven Design (DDD) is beyond the scope of this book, I'll use the concepts as a foundation for defining applicable domain classes for the example applications built in this and subsequent chapters. For a full exploration of DDD I would refer the reader to Eric Evans's seminal work on the topic, *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

By way of a cursory explanation, a *domain class* encapsulates a primary domain entity that has relevance and significance independently of other data. This doesn't mean it doesn't relate to other domain entities, only that it can stand alone and make sense as a unit, even when unassociated with other entities.

To create a domain class in Spring using Java, you can create a class with member variables, applicable constructors, accessors/mutators, and `equals()`/`hashCode()` / `toString()` methods (and more). You can also employ Lombok with Java or data classes in Kotlin to create domain classes for data representation, storage, and retrieval. I do all of these things in this chapter to demonstrate just how easy it is to work with domains using Spring Boot and Spring Data. It's great to have options.

For the examples in this chapter, once I've defined a domain class, I'll decide on a database and level of abstraction based on goals for data usage and exposed APIs for or by the database provider. Within the Spring ecosystem, this typically boils down to one of two options, with minor variations: templates or repositories.

Template Support

In order to provide a set of "just high enough" coherent abstractions, Spring Data defines an interface of type `Operations` for most of its various data sources. This `Operations` interface—examples include `MongoOperations`, `RedisOperations`, and `CassandraOperations`—specifies a foundational set of operations that can be used directly for greatest flexibility or upon which higher-level abstractions can be constructed. `Template` classes provide direct implementations of `Operations` interfaces.

Templates can be thought of as a Service Provider Interface (SPI) of sorts—directly usable and extremely capable but with many repetitive steps required each time they're used to accomplish the more common use cases developers face. For those scenarios in which data access follows common patterns, repositories may be a better choice. And the best part is that repositories build upon templates, so you lose nothing by stepping up to the higher abstraction.

Repository Support

Spring Data defines the `Repository` interface from which all other types of Spring Data repository interfaces derive. Examples include `JPARepository` and `MongoRepository` (providing JPA-specific and Mongo-specific capabilities, respectively) and more versatile interfaces like `CrudRepository`, `ReactiveCrudRepository`, and `PagingAndSortingRepository`. These various repository interfaces specify useful higher-level operations like `findAll()`, `findById()`, `count()`, `delete()`, `deleteAll()`, and more.

Repositories are defined for both blocking and nonblocking interactions. Additionally, creating queries using convention over configuration, and even literal query statements, is supported by Spring Data's repositories. Using Spring Data's repositories with Spring Boot makes building complex database interactions an almost trivial exercise.

I demonstrate all of these capabilities at some point in this book. In this chapter, I plan to cover the key elements across a number of database options by incorporating various implementation details: Lombok, Kotlin, and more. In that way, I provide a broad and stable base upon which to build in subsequent chapters.

@Before

As much as I love coffee and rely on it to drive my application development, in order to better explore the concepts covered throughout the rest of this book, I felt a more versatile domain was in order. As both a software developer and pilot, I see the increasingly complex and data-driven world of aviation as offering no shortage of interesting scenarios (and fascinating data) to explore as we delve deeper into Spring Boot's facility in numerous use cases.

To deal with data, we must *have* data. I've developed a small Spring Boot RESTful web service called `PlaneFinder` (available within this book's code repositories) to serve as an API gateway that I can poll for current aircraft and positions within range of a small device on my desk. This device receives Automatic Dependent Surveillance—Broadcast (ADS-B) data from airplanes within a certain distance and shares them with a service online, PlaneFinder.net. It also exposes an HTTP API that my gateway service consumes, simplifies, and exposes to other downstream services like the ones in this chapter.

More details throughout, but for now, let's create some database-connected services.

Creating a Template-Based Service Using Redis

Redis is a database that is typically used as an in-memory datastore for sharing state among instances of a service, caching, and brokering messages between services. Like all major databases, Redis does more, but the focus for this chapter is simply using Redis to store and retrieve from memory aircraft information our service obtains from the `PlaneFinder` service referred to previously.

Initializing the Project

To begin, we return to the Spring Initializr. From there, I choose the following options:

- Maven project
- Java
- Current production version of Spring Boot
- Packaging: Jar
- Java: 11

And for dependencies:

- Spring Reactive Web (`spring-boot-starter-webflux`)
- Spring Data Redis (Access+Driver) (`spring-boot-starter-data-redis`)
- Lombok (`lombok`)

NOTES ABOUT PROJECT OPTIONS

Artifact IDs are in parentheses after Initializr's menu name for the shown capabilities/libraries above. The first two have a common Group ID—`org.springframework.boot`—while Lombok's is `org.projectlombok`.

Although I don't specifically target any nonblocking, reactive capabilities for this chapter's applications, I include the dependency for Spring Reactive Web instead of that of Spring Web in order to gain access to `WebClient`, the preferred client for both blocking and nonblocking service interactions for applications built using Spring Boot 2.x and onward. From the perspective of a developer building a basic web service, the code is the same regardless of which dependency I include: code, annotations, and properties for this chapter's examples are fully consistent between the two. I'll point out the differences as the two paths begin to diverge in future chapters.

Next, I generate the project and save it locally, unzip it, and open it in the IDE.

Developing the Redis Service

Let's begin with the domain.

Currently, the `PlaneFinder` API gateway exposes a single REST endpoint:

```
http://localhost:7634/aircraft
```

Any (local) service can query this endpoint and receive a JSON response of all aircraft within range of the receiver in the following format (with representative data):

```
[
    {
        "id": 108,
        "callsign": "AMF4263",
        "squawk": "4136",
        "reg": "N49UC",
        "flightno": "",
        "route": "LAN-DFW",
        "type": "B190",
        "category": "A1",
        "altitude": 20000,
        "heading": 235,
        "speed": 248,
        "lat": 38.865905,
        "lon": -90.429382,
        "barometer": 0,
        "vert_rate": 0,
        "selected_altitude": 0,
        "polar_distance": 12.99378,
        "polar_bearing": 345.393951,
        "is_adsb": true,
        "is_on_ground": false,
        "last_seen_time": "2020-11-11T21:44:04Z",
        "pos_update_time": "2020-11-11T21:44:03Z",
        "bds40_seen_time": null
    },
    {<another aircraft in range, same fields as above>},
    {<final aircraft currently in range, same fields as above>}
]
```

Defining the domain class

In order to ingest and manipulate these aircraft reports, I create an `Aircraft` class as follows:

```
package com.thehecklers.sburredis;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;

import java.time.Instant;

@Data
@NoArgsConstructor
@AllArgsConstructor
@JsonIgnoreProperties(ignoreUnknown = true)
public class Aircraft {
    @Id
    private Long id;
    private String callsign, squawk, reg, flightno, route, type, category;
    private int altitude, heading, speed;
    @JsonProperty("vert_rate")
    private int vertRate;
    @JsonProperty("selected_altitude")
    private int selectedAltitude;
    private double lat, lon, barometer;
    @JsonProperty("polar_distance")
```

```java
    private double polarDistance;
    @JsonProperty("polar_bearing")
    private double polarBearing;
    @JsonProperty("is_adsb")
    private boolean isADSB;
    @JsonProperty("is_on_ground")
    private boolean isOnGround;
    @JsonProperty("last_seen_time")
    private Instant lastSeenTime;
    @JsonProperty("pos_update_time")
    private Instant posUpdateTime;
    @JsonProperty("bds40_seen_time")
    private Instant bds40SeenTime;

    public String getLastSeenTime() {
        return lastSeenTime.toString();
    }

    public void setLastSeenTime(String lastSeenTime) {
        if (null != lastSeenTime) {
            this.lastSeenTime = Instant.parse(lastSeenTime);
        } else {
            this.lastSeenTime = Instant.ofEpochSecond(0);
        }
    }

    public String getPosUpdateTime() {
        return posUpdateTime.toString();
    }

    public void setPosUpdateTime(String posUpdateTime) {
        if (null != posUpdateTime) {
            this.posUpdateTime = Instant.parse(posUpdateTime);
        } else {
            this.posUpdateTime = Instant.ofEpochSecond(0);
        }
    }

    public String getBds40SeenTime() {
        return bds40SeenTime.toString();
    }

    public void setBds40SeenTime(String bds40SeenTime) {
        if (null != bds40SeenTime) {
            this.bds40SeenTime = Instant.parse(bds40SeenTime);
        } else {
            this.bds40SeenTime = Instant.ofEpochSecond(0);
        }
    }
}
```

This domain class includes a few helpful annotations that streamline the necessary code and/or increase its flexibility. Class-level annotations include the following:

`@Data` :: Instructs Lombok to create getter, setter, `equals()`, `hashCode()`, and `toString()` methods, creating a so-called data class `@NoArgsConstructor` :: Instructs Lombok to create a zero-parameter constructor, thus requiring no arguments `@AllArgsConstructor` :: Instructs Lombok to create a constructor with a parameter for each member variable, requiring an argument be provided for all `@JsonIgnoreProperties(ignoreUnknown = true)` :: Informs Jackson deserialization mechanisms to ignore fields within JSON responses for which there is no corresponding member variable

Field-level annotations provide more specific guidance where appropriate. Examples of field-level annotations include the two used for this class:

`@Id` :: Designates the annotated member variable as holding the unique identifier for a database entry/record `@JsonProperty("vert_rate")` :: Connects a member variable with its differently named JSON field

You may be wondering why I created explicit accesssors and mutators for the three member variables of type `Instant` if the `@Data` annotation results in the creation of getter and setter methods for all member variables. In the case of these three, the JSON value must be parsed and transformed from a `String` to a complex data type by calling a method: `Instant::parse`. If that value is entirely absent (null), different logic must be performed to avoid passing a null to `parse()` and to assign some meaningful substitute value to the corresponding member variable via setter. Additionally, serialization of `Instant` values is best done by conversion to a `String`—thus the explicit getter methods.

With a domain class defined, it’s time to create and configure the mechanism for accessing a Redis database.

Adding template support

Spring Boot provides basic `RedisTemplate` capabilities via autoconfiguration, and if you only need to manipulate `String` values using Redis, very little work (or code) is required from you. Dealing with complex domain objects necessitates a bit more configuration but not too much.

The `RedisTemplate` class extends the `RedisAccessor` class and implements the `RedisOperations` interface. Of particular interest for this application is `RedisOperations`, as it specifies the functionality needed to interact with Redis.

As developers, we should prefer to write code against interfaces, not implementations. Doing so allows one to provide the most appropriate concrete implementation for the task at hand without code/API changes or excessive and unnecessary violations of the DRY (Don't Repeat Yourself) principle; as long as the interface is fully implemented, any concrete implementation will function just as well as any other.

In the following code listing, I create a bean of type `RedisOperations`, returning a `RedisTemplate` as the bean's concrete implementation. I perform the following steps in order to configure it properly to accommodate inbound `Aircraft`:

1. I create a `Serializer` to be used when converting between objects and JSON records. Since Jackson is used for marshalling/unmarshalling (serialization/deserialization) of JSON values and is already present in Spring Boot web applications, I create a `Jackson2JsonRedisSerializer` for objects of type `Aircraft`.
2. I create a `RedisTemplate` that accepts keys of type `String` and values of type `Aircraft` to accommodate the inbound `Aircraft` with `String` IDs. . I plug the `RedisConnectionFactory` bean that was helpfully and automatically autowired into this bean-creation method's sole parameter—`RedisConnectionFactory factory`—into the `template` object so it can create and retrieve a connection to the Redis database.
3. I supply the `Jackson2JsonRedisSerializer<Aircraft>` serializer to the `template` object in order to be used as the default serializer. `RedisTemplate` has a number of serializers that are assigned the default serializer in the absence of specific assignment, a useful touch.
4. I create and specify a different serializer to be used for keys so that the template doesn't attempt to use the default serializer—which expects objects of type `Aircraft`—to convert to/from key values of type `String`. A `StringRedisSerializer` does the trick nicely.
5. Finally, I return the created and configured `RedisTemplate` as the bean to use when some implementation of a `RedisOperations` bean is requested within the application:

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisOperations;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerialize
import org.springframework.data.redis.serializer.StringRedisSerializer;

@SpringBootApplication
public class SburRedisApplication {
    @Bean
    public RedisOperations<String, Aircraft>
    redisOperations(RedisConnectionFactory factory) {
        Jackson2JsonRedisSerializer<Aircraft> serializer =
                new Jackson2JsonRedisSerializer<>(Aircraft.class);

        RedisTemplate<String, Aircraft> template = new RedisTemplate<>();
        template.setConnectionFactory(factory);
        template.setDefaultSerializer(serializer);
        template.setKeySerializer(new StringRedisSerializer());

        return template;
    }

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

Bringing it all together

Now that the underlying wiring is in place for accessing the Redis database using a template, it's time for the payoff. As shown in the code listing that follows, I create a Spring Boot `@Component` class to poll the `PlaneFinder` endpoint and handle the resultant `Aircraft` records it receives using Redis template support.

To initialize the `PlaneFinderPoller` bean and prepare it for action, I create a `WebClient` object and assign it to a member variable, pointing it to the destination endpoint exposed by the external `PlaneFinder` service. `PlaneFinder` currently runs on my local machine and listens on port 7634.

The `PlaneFinderPoller` bean requires access to two other beans to perform its duties: a `RedisConnectionFactory` (supplied by Boot's autoconfiguration due to Redis being an app dependency) and an implementation of `RedisOperations`, the `RedisTemplate` created earlier. Both are assigned to properly defined member variables via constructor injection (autowired):

```java
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisOperations;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@EnableScheduling
@Component
class PlaneFinderPoller {
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");

    private final RedisConnectionFactory connectionFactory;
    private final RedisOperations<String, Aircraft> redisOperations;

    PlaneFinderPoller(RedisConnectionFactory connectionFactory,
                      RedisOperations<String, Aircraft> redisOperations) {
        this.connectionFactory = connectionFactory;
        this.redisOperations = redisOperations;
    }
}
```

Next, I create the method that does the heavy lifting. In order to have it poll on a fixed schedule, I leverage the `@EnableScheduling` annotation I previously placed at the class level and annotate the `pollPlanes()` method I create with `@Scheduled`, supplying a parameter of `fixedDelay=1000` to specify a polling frequency of once per 1,000 ms—once per second. The rest of the method consists of only three declarative statements: one to clear any previously saved `Aircraft`, one to retrieve and save current positions, and one to report the results of the latest capture.

For the first task I use the autowired `ConnectionFactory` to obtain a connection to the database, and via that connection, I execute the server command to clear all keys present: `flushDb()`.

The second statement uses the `WebClient` to call the `PlaneFinder` service and retrieve a collection of aircraft within range, along with their current position information. The response body is converted to a `Flux` of `Aircraft` objects, filtered to remove any `Aircraft` that don't include registration numbers, converted to a `Stream` of `Aircraft`, and saved to the Redis database. The save is performed on each valid `Aircraft` by setting a key/value pair to the `Aircraft` registration number and the `Aircraft` object itself, respectively, using Redis's operations tailored to manipulating data values.

NOTE

A `Flux` is a reactive type covered in upcoming chapters, but for now, simply think of it as a collection of objects delivered without blocking.

The final statement in `pollPlanes()` again leverages a couple of Redis's defined value operations to retrieve all keys (via the wildcard parameter *) and, using each key, to retrieve each corresponding `Aircraft` value, which is then printed. Here is the `pollPlanes()` method in finished form:

```
@Scheduled(fixedRate = 1000)
private void pollPlanes() {
    connectionFactory.getConnection().serverCommands().flushDb();

    client.get()
            .retrieve()
            .bodyToFlux(Aircraft.class)
            .filter(plane -> !plane.getReg().isEmpty())
            .toStream()
            .forEach(ac -> redisOperations.opsForValue().set(ac.getReg(), a

    redisOperations.opsForValue()
            .getOperations()
            .keys("*")
            .forEach(ac ->
                System.out.println(redisOperations.opsForValue().get(ac)));
}
```

The final version (for now) of the `PlaneFinderPoller` class is shown in the following listing:

```
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisOperations;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@EnableScheduling
@Component
class PlaneFinderPoller {
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");

    private final RedisConnectionFactory connectionFactory;
    private final RedisOperations<String, Aircraft> redisOperations;
    PlaneFinderPoller(RedisConnectionFactory connectionFactory,
                      RedisOperations<String, Aircraft> redisOperations) {
        this.connectionFactory = connectionFactory;
        this.redisOperations = redisOperations;
    }

    @Scheduled(fixedRate = 1000)
    private void pollPlanes() {
        connectionFactory.getConnection().serverCommands().flushDb();

        client.get()
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(plane -> !plane.getReg().isEmpty())
                .toStream()
                .forEach(ac ->
                    redisOperations.opsForValue().set(ac.getReg(), ac));

        redisOperations.opsForValue()
                .getOperations()
                .keys("*")
                .forEach(ac ->
                    System.out.println(redisOperations.opsForValue().get(ac
    }
}
```

With polling mechanisms fully fleshed out, let's run the application and see the results.

The results

With the `PlaneFinder` service already running on my machine, I start the *sbur-redis* application to obtain, store and retrieve in Redis, and display the results of each poll of `PlaneFinder`. What follows is an example of the results, edited for brevity and formatted a bit for readability:

```
Aircraft(id=1, callsign=EDV5015, squawk=3656, reg=N324PQ, flightno=DL5015,
route=ATL-OMA-ATL, type=CRJ9, category=A3, altitude=35000, heading=168,
speed=485, vertRate=-64, selectedAltitude=0, lat=38.061808, lon=-90.280629,
barometer=0.0, polarDistance=53.679699, polarBearing=184.333345, isADSB=tru
isOnGround=false, lastSeenTime=2020-11-27T18:34:14Z,
posUpdateTime=2020-11-27T18:34:11Z, bds40SeenTime=1970-01-01T00:00:00Z)

Aircraft(id=4, callsign=AAL500, squawk=2666, reg=N839AW, flightno=AA500,
route=PHX-IND, type=A319, category=A3, altitude=36975, heading=82, speed=47
vertRate=0, selectedAltitude=36992, lat=38.746399, lon=-90.277644,
barometer=1012.8, polarDistance=13.281347, polarBearing=200.308663, isADSB=
isOnGround=false, lastSeenTime=2020-11-27T18:34:50Z,
posUpdateTime=2020-11-27T18:34:50Z, bds40SeenTime=2020-11-27T18:34:50Z)
```

```
Aircraft(id=15, callsign=null, squawk=4166, reg=N404AN, flightno=AA685,
route=PHX-DCA, type=A21N, category=A3, altitude=39000, heading=86, speed=49
vertRate=0, selectedAltitude=39008, lat=39.701611, lon=-90.479309,
barometer=1013.6, polarDistance=47.113195, polarBearing=341.51817, isADSB=t
isOnGround=false, lastSeenTime=2020-11-27T18:34:50Z,
posUpdateTime=2020-11-27T18:34:50Z, bds40SeenTime=2020-11-27T18:34:50Z)
```

Working with databases via Spring Data's template support provides a lower-level API with excellent flexibility. If you're looking for minimum friction and maximum productivity and repeatability, however, repository support is the better choice. Next, I show how to convert from using templates to interact with Redis to using a Spring Data repository. It's great to have options.

Converting from Template to Repository

Before we can use a repository, it's necessary to define one, and Spring Boot's autoconfiguration helps considerably with this. I create a repository interface as follows, extending Spring Data's `CrudRepository` and providing the type of object to store along with its key: `Aircraft` and `Long`, in this case:

```
public interface AircraftRepository extends CrudRepository<Aircraft, Long>
```

As explained in Chapter 4, Spring Boot detects the Redis database driver on the application classpath and notes that we're extending a Spring Data repository interface, then creates a database proxy automatically with no additional code required to instantiate it. Just like that, the application has access to an `AircraftRepository` bean. Let's plug it in and put it to use.

Revisiting the `PlaneFinderPoller` class, I can now replace the lower-level references to and operations using `RedisOperations` and replace them with `AircraftRepository`.

First, I remove the `RedisOperations` member variable:

```
private final RedisOperations<String, Aircraft> redisOperations;
```

Then replace it with one for the `AircraftRepository` to autowire:

```
private final AircraftRepository repository;
```

Next, I replace the `RedisOperations` bean autowired via constructor injection with the `AircraftRepository` and the assignment within the constructor to the applicable member variable so that the constructor ends up like so:

```
public PlaneFinderPoller(RedisConnectionFactory connectionFactory,
                         AircraftRepository repository) {
    this.connectionFactory = connectionFactory;
    this.repository = repository;
}
```

The next step is to refactor the `pollPlanes()` method to replace template-based operations with repository-based ops.

Changing the last line of the first statement is a simple matter. Using a method reference further simplifies the lambda:

```
client.get()
        .retrieve()
        .bodyToFlux(Aircraft.class)
        .filter(plane -> !plane.getReg().isEmpty())
        .toStream()
        .forEach(repository::save);
```

And the second one reduces even more, again including use of a method reference:

```
repository.findAll().forEach(System.out::println);
```

The newly repository-enabled `PlaneFinderPoller` now consists of the following code:

```
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@EnableScheduling
@Component
class PlaneFinderPoller {
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");
    private final RedisConnectionFactory connectionFactory;
    private final AircraftRepository repository;

    PlaneFinderPoller(RedisConnectionFactory connectionFactory,
                      AircraftRepository repository) {
        this.connectionFactory = connectionFactory;
        this.repository = repository;
    }

    @Scheduled(fixedRate = 1000)
    private void pollPlanes() {
        connectionFactory.getConnection().serverCommands().flushDb();

        client.get()
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(plane -> !plane.getReg().isEmpty())
                .toStream()
                .forEach(repository::save);

        repository.findAll().forEach(System.out::println);
    }
}
```

With no further need of a bean implementing the `RedisOperations` interface, I can now delete its `@Bean` definition from the main application class, leaving `SburRedisApplication`, as shown in the following code:

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

@SpringBootApplication
public class SburRedisApplication {

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

}
```

Only one small task and a very nice code reduction remain to fully enable Redis repository support in our application. I add the `@RedisHash` annotation to the `Aircraft` entity to indicate that `Aircraft` is an aggregate root to be stored in a Redis hash, performing a function similar to what `@Entity` annotation does for JPA objects. I then remove the explicit accessors and mutators previously required for the `Instant`-typed member variables, as the converters in Spring Data's repository support handle complex type conversions with ease. The newly streamlined `Aircraft` class now looks like this:

```java
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.redis.core.RedisHash;

import java.time.Instant;

@Data
@NoArgsConstructor
@AllArgsConstructor
@RedisHash
@JsonIgnoreProperties(ignoreUnknown = true)
public class Aircraft {
    @Id
    private Long id;
    private String callsign, squawk, reg, flightno, route, type, category;
    private int altitude, heading, speed;
    @JsonProperty("vert_rate")
    private int vertRate;
    @JsonProperty("selected_altitude")
    private int selectedAltitude;
    private double lat, lon, barometer;
    @JsonProperty("polar_distance")
    private double polarDistance;
    @JsonProperty("polar_bearing")
    private double polarBearing;
    @JsonProperty("is_adsb")
    private boolean isADSB;
    @JsonProperty("is_on_ground")
    private boolean isOnGround;
    @JsonProperty("last_seen_time")
    private Instant lastSeenTime;
    @JsonProperty("pos_update_time")
    private Instant posUpdateTime;
    @JsonProperty("bds40_seen_time")
    private Instant bds40SeenTime;
}
```

With the latest changes in place, restarting the service results in output indistinguishable from the template-based approach but with much less code and inherent ceremony required. An example of results follows, again edited for brevity and formatted for readability:

```
Aircraft(id=59, callsign=KAP20, squawk=4615, reg=N678JG, flightno=,
route=STL-IRK, type=C402, category=A1, altitude=3825, heading=0, speed=143,
vertRate=768, selectedAltitude=0, lat=38.881034, lon=-90.261475, barometer=
polarDistance=5.915421, polarBearing=222.434158, isADSB=true, isOnGround=fa
lastSeenTime=2020-11-27T18:47:31Z, posUpdateTime=2020-11-27T18:47:31Z,
bds40SeenTime=1970-01-01T00:00:00Z)

Aircraft(id=60, callsign=SWA442, squawk=5657, reg=N928WN, flightno=WN442,
route=CMH-DCA-BNA-STL-PHX-BUR-OAK, type=B737, category=A3, altitude=8250,
heading=322, speed=266, vertRate=-1344, selectedAltitude=0, lat=38.604034,
lon=-90.357593, barometer=0.0, polarDistance=22.602864, polarBearing=201.28
isADSB=true, isOnGround=false, lastSeenTime=2020-11-27T18:47:25Z,
posUpdateTime=2020-11-27T18:47:24Z, bds40SeenTime=1970-01-01T00:00:00Z)

Aircraft(id=61, callsign=null, squawk=null, reg=N702QS, flightno=,
route=SNA-RIC, type=CL35, category=, altitude=43000, heading=90, speed=500,
vertRate=0, selectedAltitude=0, lat=39.587997, lon=-90.921299, barometer=0.
polarDistance=51.544552, polarBearing=316.694343, isADSB=true, isOnGround=f
lastSeenTime=2020-11-27T18:47:19Z, posUpdateTime=2020-11-27T18:47:19Z,
bds40SeenTime=1970-01-01T00:00:00Z)
```

If you need direct access to the lower-level capabilities exposed by Spring Data templates, template-based database support is indispensable. But for nearly all common use cases, when Spring Data offers repository-based access for a target database, it's best to begin—and in all likelihood remain—there.

Creating a Repository-Based Service Using the Java Persistence API (JPA)

One of the Spring ecosystem's strengths is consistency: once you learn how to accomplish something, the same approach can be applied to drive successful outcomes with different components. Database access is a case in point.

Spring Boot and Spring Data provide repository support for a number of different databases: JPA-compliant databases, numerous NoSQL datastores of varying types, and in-memory and/or persistent stores. Spring smooths the bumps a developer runs into when transitioning between databases, whether for a single application or throughout a vast system of them.

To demonstrate some of the flexible options at your disposal when creating data-aware Spring Boot applications, I highlight a few different approaches supported by Spring Boot in each of the following sections, while relying on Boot (and Spring Data) to streamline the database portion of the different, but similar, services. First up is JPA, and for this example I use Lombok throughout to reduce code and increase readability.

Initializing the Project

Once again we return to the Spring Initializr. This time, I choose the following options:

- Maven project
- Java
- Current production version of Spring Boot
- Packaging: Jar
- Java: 11

And for dependencies:

- Spring Reactive Web (`spring-boot-starter-webflux`)
- Spring Data JPA (`spring-boot-starter-data-jpa`)
- MySQL Driver (`mysql-connector-java`)
- Lombok (`lombok`)

Next, I generate the project and save it locally, unzip it, and open it in the IDE.

NOTE

As with the earlier Redis project and most other examples in this chapter, each data-aware service must be able to access a running database. Please refer to this book's associated code repositories for Docker scripts to create and run suitable containerized database engines.

Developing the JPA (MySQL) Service

Considering both Chapter 4's example built using JPA and the H2 database and the previous Redis repository-based example, the JPA-based service using MariaDB/MySQL clearly demonstrates the way in which Spring's consistency amplifies developer productivity.

Defining the domain class

As with all of this chapter's projects, I create an `Aircraft` domain class to serve as the primary (data) focus. Each different project will have slight variations around a common theme pointed out along the way. Here is the JPA-centric `Aircraft` domain class structure:

```java
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.time.Instant;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Aircraft {
    @Id
    @GeneratedValue
    private Long id;

    private String callsign, squawk, reg, flightno, route, type, category;

    private int altitude, heading, speed;
    @JsonProperty("vert_rate")
    private int vertRate;
    @JsonProperty("selected_altitude")
    private int selectedAltitude;

    private double lat, lon, barometer;
    @JsonProperty("polar_distance")
    private double polarDistance;
    @JsonProperty("polar_bearing")
    private double polarBearing;

    @JsonProperty("is_adsb")
    private boolean isADSB;
    @JsonProperty("is_on_ground")
    private boolean isOnGround;

    @JsonProperty("last_seen_time")
    private Instant lastSeenTime;
    @JsonProperty("pos_update_time")
    private Instant posUpdateTime;
    @JsonProperty("bds40_seen_time")
    private Instant bds40SeenTime;
}
```

There are a few particulars of note with regard to this version of `Aircraft` versus prior versions and those to come.

First, the `@Entity`, `@Id`, and `@GeneratedValue` annotations are all imported from the `javax.persistence` package. You may remember that in the Redis version (and some others), `@Id` comes from `org.springframework.data.annotation`.

Class-level annotations closely parallel those used in the example using Redis repository support, with the replacement of `@RedisHash` with a JPA `@Entity` annotation. To revisit the other (unchanged) annotations shown, please refer to the aforementioned earlier section.

Field-level annotations are also similar, with the addition of `@GeneratedValue`. As its name implies, `@GeneratedValue` indicates that the identifier will be generated by the underlying database engine. The developer can—if desired or necessary—provide additional guidance for key generation, but for our purposes, the annotation itself is sufficient.

As with Spring Data's repository support for Redis, there is no need for explicit accessors/mutators for the member variables of type `Instant`, leaving (once again) a very svelte `Aircraft` domain class.

Creating the repository interface

Next, I define the required repository interface, extending Spring Data's `CrudRepository` and providing the type of object to store and its key: `Aircraft` and `Long`, in this case:

```
public interface AircraftRepository extends CrudRepository<Aircraft, Long>
```

NOTE

Both Redis and JPA databases function well with unique key values/identifiers of type `Long`, so this is identical to the one defined in the earlier Redis example.

Bringing it all together

Now to create the `PlaneFinder` polling component and configure it for database access.

Polling PlaneFinder

Once again I create a Spring Boot `@Component` class to poll for current position data and handle the resultant `Aircraft` records it receives.

Like the earlier example, I create a `WebClient` object and assign it to a member variable, pointing it to the destination endpoint exposed by the `PlaneFinder` service on port 7634.

As you should expect from a sibling repository implementation, the code is quite similar to the Redis repository endstate. I demonstrate a couple of differences in approach for this example.

Rather than manually creating a constructor via which to receive the autowired `AircraftRepository` bean, I instruct Lombok—via its compile-time code generator—to provide a constructor with any required member variables. Lombok determines which arguments are required via two annotations: `@RequiredArgsConstructor` on the class and `@NonNull` on the member variable(s) designated as requiring initialization. By annotating the `AircraftRepository` member variable as an `@NonNull` property, Lombok creates a constructor with an `AircraftRepository` as a parameter; Spring Boot then dutifully autowires the existing repository bean for use within the `PlaneFinderPoller` bean.

NOTE

The wisdom of deleting all stored entries in a database each time a poll is conducted depends heavily on requirements, polling frequency, and storage mechanism involved. For example, the costs involved in clearing an in-memory database before each poll is quite different from deleting all records in a cloud-hosted database's table. Frequent polling also increases associated costs. Alternatives exist; please choose wisely.

To revisit the details of the remaining code in `PlaneFinderPoller`, please review the corresponding section under Redis repository support. Refactored to take full advantage of Spring Data JPA support, the complete code for `PlaneFinderPoller` is shown in the following listing:

```java
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@EnableScheduling
@Component
@RequiredArgsConstructor
class PlaneFinderPoller {
    @NonNull
    private final AircraftRepository repository;
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");

    @Scheduled(fixedRate = 1000)
    private void pollPlanes() {
        repository.deleteAll();

        client.get()
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(plane -> !plane.getReg().isEmpty())
                .toStream()
                .forEach(repository::save);

        repository.findAll().forEach(System.out::println);
    }
}
```

Connecting to MariaDB/MySQL

Spring Boot autoconfigures the application’s environment using all information available at runtime; that’s one of the key enablers of its unrivaled flexibility. Since there are many JPA-compliant databases supported by Spring Boot and Spring Data, we need to provide a few key bits of information for Boot to use to seamlessly connect to the database of our choosing for this particular application. For this service running in my environment, these properties include:

```
spring.datasource.platform=mysql
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/mark
spring.datasource.username=mark
spring.datasource.password=sbux
```

NOTE

Both the database name and the database username are “mark” in the example above. Replace datasource, username, and password values with those specific to your environment.

The results

With the `PlaneFinder` service still running on my machine, I start the *sbur-jpa* service to obtain, store and retrieve (in MariaDB), and display the results of each polling of `PlaneFinder`. An example of the results follows, edited for brevity and formatted for readability:

```
Aircraft(id=106, callsign=null, squawk=null, reg=N7816B, flightno=WN2117,
route=SJC-STL-BWI-FLL, type=B737, category=, altitude=4400, heading=87,
speed=233, vertRate=2048, selectedAltitude=15008, lat=0.0, lon=0.0,
barometer=1017.6, polarDistance=0.0, polarBearing=0.0, isADSB=false,
isOnGround=false, lastSeenTime=2020-11-27T18:59:10Z,
posUpdateTime=2020-11-27T18:59:17Z, bds40SeenTime=2020-11-27T18:59:10Z)

Aircraft(id=107, callsign=null, squawk=null, reg=N963WN, flightno=WN851,
route=LAS-DAL-STL-CMH, type=B737, category=, altitude=27200, heading=80,
speed=429, vertRate=2112, selectedAltitude=0, lat=0.0, lon=0.0, barometer=0
polarDistance=0.0, polarBearing=0.0, isADSB=false, isOnGround=false,
lastSeenTime=2020-11-27T18:58:45Z, posUpdateTime=2020-11-27T18:59:17Z,
bds40SeenTime=2020-11-27T18:59:17Z)

Aircraft(id=108, callsign=null, squawk=null, reg=N8563Z, flightno=WN1386,
route=DEN-IAD, type=B738, category=, altitude=39000, heading=94, speed=500,
vertRate=0, selectedAltitude=39008, lat=0.0, lon=0.0, barometer=1013.6,
polarDistance=0.0, polarBearing=0.0, isADSB=false, isOnGround=false,
lastSeenTime=2020-11-27T18:59:10Z, posUpdateTime=2020-11-27T18:59:17Z,
bds40SeenTime=2020-11-27T18:59:10Z)
```

The service works as expected to poll, capture, and display aircraft positions.

Loading Data

This chapter's focus thus far has been how to interact with a database when data flows into the application. What happens if data exists—sample, test, or actual seed data—that must be persisted?

Spring Boot has a few different mechanisms to initialize and populate a database. I cover what I consider to be the two most useful approaches here:

- Using Data Definition Language (DDL) and Data Manipulation Language (DML) scripts to initialize and populate
- Allowing Boot (via Hibernate) to automatically create the table structure from defined `@Entity` class(es) and populating via a repository bean

Each approach to data definition and population has its pros and cons.

API- or database-specific scripts

Spring Boot checks the usual root classpath locations for files that fit the following naming format:

- *schema.sql*
- *data.sql*
- *schema-${platform}.sql*
- *data-${platform}.sql*

The last two filenames are matched to the developer-assigned application property `spring.datasource.platform`. Valid values include `h2`, `mysql`, `postgresql`, and other Spring Data JPA databases, and using a combination of the `spring.datasource.platform` property and related `.sql` files enables a developer to fully leverage syntax specific to that particular database.

Creating and populating with scripts

To leverage scripts to create and populate a MariaDB/MySQL database in the most straightforward way, I create two files under the `resources` directory of the *sbur-jpa* project: *schema-mysql.sql* and *data-mysql.sql*.

To create the `aircraft` table schema, I add the following DDL to *schema-mysql.sql*:

```
DROP TABLE IF EXISTS aircraft;
CREATE TABLE aircraft (id BIGINT not null primary key, callsign VARCHAR(7),
squawk VARCHAR(4), reg VARCHAR(6), flightno VARCHAR(10), route VARCHAR(25),
type VARCHAR(4), category VARCHAR(2),
altitude INT, heading INT, speed INT, vert_rate INT, selected_altitude INT,
lat DOUBLE, lon DOUBLE, barometer DOUBLE,
polar_distance DOUBLE, polar_bearing DOUBLE,
isadsb BOOLEAN, is_on_ground BOOLEAN,
last_seen_time TIMESTAMP, pos_update_time TIMESTAMP, bds40seen_time TIMESTA
```

To populate the `aircraft` table with a single sample row, I add the following DML to *data-mysql.sql*:

```
INSERT INTO aircraft (id, callsign, squawk, reg, flightno, route, type,
category, altitude, heading, speed, vert_rate, selected_altitude, lat, lon,
barometer, polar_distance, polar_bearing, isadsb, is_on_ground,
last_seen_time, pos_update_time, bds40seen_time)
VALUES (81, 'AAL608', '1451', 'N754UW', 'AA608', 'IND-PHX', 'A319', 'A3', 3
255, 423, 0, 36000, 39.150284, -90.684795, 1012.8, 26.575562, 295.501994,
true, false, '2020-11-27 21:29:35', '2020-11-27 21:29:34',
'2020-11-27 21:29:27');
```

By default, Boot automatically creates table structures from any classes annotated with `@Entity`. It's simple to override this behavior with the following property settings, shown here from the app's *application.properties* file:

```
spring.datasource.initialization-mode=always
spring.jpa.hibernate.ddl-auto=none
```

Setting `spring.datasource.initialization-mode` to “always” indicates that the app is expecting to use an external (nonembedded) database and should initialize it each time the application executes. Setting `spring.jpa.hibernate.ddl-auto` to “none” disables Spring Boot's automatic table creation from `@Entity` classes.

To verify that the preceding scripts are being used to create and populate the `aircraft` table, I visit the `PlaneFinderPoller` class and do the following:

- Comment out the `repository.deleteAll();` statement in `pollPlanes()`. This is necessary to avoid deleting the record added via *data-mysql.sql*.
- Comment out the `client.get()...` statement, also in `pollPlanes()`. This results in no additional records being retrieved and created from polling the external `PlaneFinder` service for easier verification.

Restarting the *sbur-jpa* service now results in the following output (`id` fields may differ), edited for brevity and formatted for clarity:

```
Aircraft(id=81, callsign=AAL608, squawk=1451, reg=N754UW, flightno=AA608,
route=IND-PHX, type=A319, category=A3, altitude=36000, heading=255, speed=4
vertRate=0, selectedAltitude=36000, lat=39.150284, lon=-90.684795,
barometer=1012.8, polarDistance=26.575562, polarBearing=295.501994, isADSB=
isOnGround=false, lastSeenTime=2020-11-27T21:29:35Z,
posUpdateTime=2020-11-27T21:29:34Z, bds40SeenTime=2020-11-27T21:29:27Z)
```

NOTE

The only record saved is the one specified in *data-mysql.sql*.

Like all approaches to anything, there are pros and cons to this method of table creation and population. Upsides include:

- The ability to directly use SQL scripts, both DDL and DML, leveraging existing scripts and/or SQL expertise
- Access to SQL syntax specific to the chosen database

Downsides aren't particularly serious but should be recognized:

- Using SQL files is obviously specific to SQL-supporting relational databases.
- Scripts can rely on SQL syntax for a particular database, which can require editing if the choice of underlying database changes.
- Some (two) application properties must be set to override default Boot behavior.

Populating the database using the application's repository

There is another way, one that I find particularly powerful and more flexible: using Boot's default behavior to create the table structures (if they don't already exist) and the application's repository support to populate sample data.

To restore Spring Boot's default behavior of creating the `aircraft` table from the `Aircraft` JPA `@Entity` class, I comment out the two properties just added to *application.properties*:

```
#spring.datasource.initialization-mode=always
#spring.jpa.hibernate.ddl-auto=none
```

With these properties no longer being defined, Spring Boot will not search for and execute *data-mysql.sql* or other data initialization scripts.

Next, I create a class with a purpose-descriptive name like `DataLoader`. I add class-level annotations of `@Component` (so Spring creates a `DataLoader` bean) and `@AllArgsConstructor` (so Lombok creates a constructor with a parameter for each member variable). I then add a single member variable to hold the `AircraftRepository` bean Spring Boot will autowire for me via constructor injection:

```
private final AircraftRepository repository;
```

And a method called `loadData()` to both clear and populate the `aircraft` table:

```
@PostConstruct
private void loadData() {
    repository.deleteAll();

    repository.save(new Aircraft(81L,
            "AAL608", "1451", "N754UW", "AA608", "IND-PHX", "A319", "A3",
            36000, 255, 423, 0, 36000,
            39.150284, -90.684795, 1012.8, 26.575562, 295.501994,
            true, false,
            Instant.parse("2020-11-27T21:29:35Z"),
            Instant.parse("2020-11-27T21:29:34Z"),
            Instant.parse("2020-11-27T21:29:27Z")));
}
```

And that's it. Really. Restarting the the *sbur-jpa* service now results in the output that follows (`id` fields may differ), edited for brevity and formatted for clarity:

```
Aircraft(id=110, callsign=AAL608, squawk=1451, reg=N754UW, flightno=AA608,
route=IND-PHX, type=A319, category=A3, altitude=36000, heading=255, speed=4
vertRate=0, selectedAltitude=36000, lat=39.150284, lon=-90.684795,
barometer=1012.8, polarDistance=26.575562, polarBearing=295.501994, isADSB=
isOnGround=false, lastSeenTime=2020-11-27T21:29:35Z,
posUpdateTime=2020-11-27T21:29:34Z, bds40SeenTime=2020-11-27T21:29:27Z)
```

NOTE

The only record saved is the one defined in the previous `DataLoader` class, with one small difference: since the `id` field is generated by the database (as specified in the `Aircraft` domain class specification), the provided `id` value is replaced by the database engine when the record is saved.

Advantages of this approach are significant:

- Fully database independent.
- Any code/annotations specific to a particular database are already within the app simply to support db access.
- Easy to disable by simply commenting out the `@Component` annotation on the `DataLoader` class.

Other mechanisms

These are two powerful and widely used options for database initialization and population, but there are other options, including using Hibernate support for an *import.sql* file (similar to the JPA approach introduced earlier), using external imports, and using FlywayDB, among others. Exploring the numerous other options is out of scope for this book and is left as an optional exercise for the reader.

Creating a Repository-Based Service Using a NoSQL Document Database

As mentioned earlier, there are several ways to further enhance developer productivity when creating applications using Spring Boot. One of these is to increase code conciseness by using Kotlin as the foundational app language.

An exhaustive exploration of the Kotlin language is well beyond the scope of this book, and there are other books that fulfill that role. Fortunately, though, while Kotlin definitely differs from Java in numerous meaningful ways, it is similar enough to pose no great hardship in adapting to its idioms with a few well-placed explanations when things diverge from the "Java way." I'll endeavor to provide those explanations as I proceed; for background or additional information, please refer to Kotlin-specific tomes.

For this example, I use MongoDB. Perhaps the best-known document datastore, MongoDB is widely used and wildly popular for good reason: it works well and generally makes life easier for developers to store, manipulate, and retrieve data in all of its varied (and sometimes messy) forms. The team at MongoDB also constantly strives to improve their feature set, security, and APIs: MongoDB was one of the first databases to offer reactive database drivers, leading the industry in taking nonblocking access all the way down to the database level.

Initializing the Project

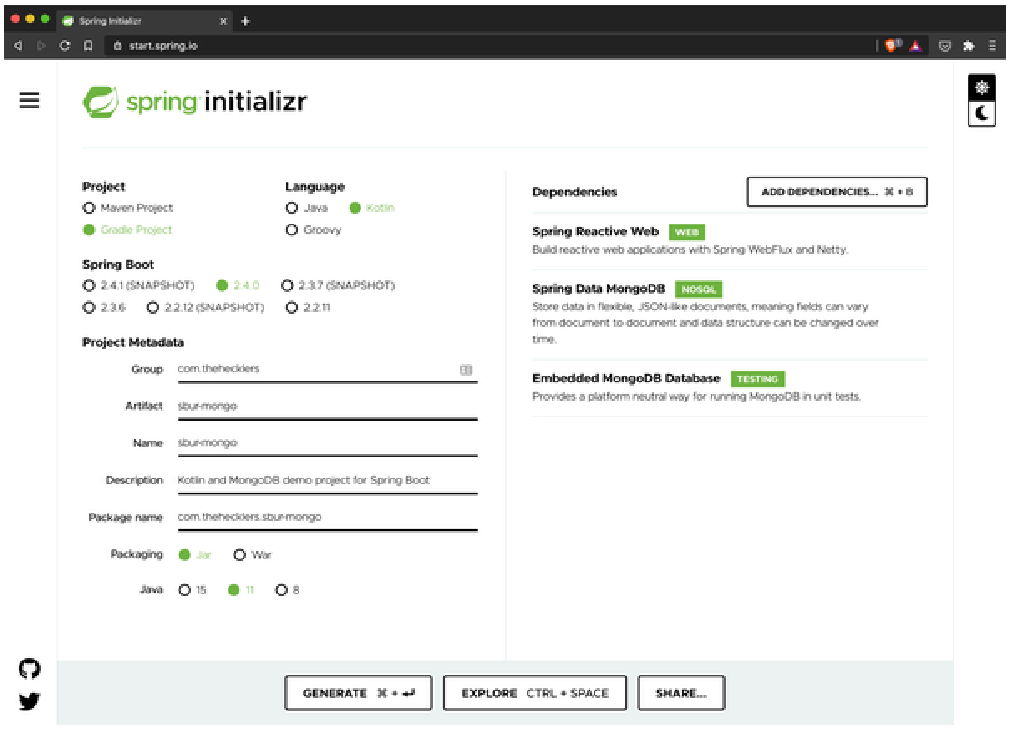
As you might expect, we return to the Spring Initializr to get started. For this project, I choose the following options (also shown in Figure 6-1)—somewhat of a departure from prior visits:

- Gradle project
- Kotlin
- Current production version of Spring Boot
- Packaging: Jar
- Java: 11

And for dependencies:

- Spring Reactive Web (`spring-boot-starter-webflux`)
- Spring Data MongoDB (`spring-boot-starter-data-mongodb`)
- Embedded MongoDB Database (`de.flapdoodle.embed.mongo`)

Next, I generate the project and save it locally, unzip it, and open it in the IDE.



Project: Maven Project [✓ Gradle Project]
Language: Java [✓ Kotlin] Groovy
Spring Boot: 2.4.1 (SNAPSHOT) [✓ 2.4.0] 2.3.7 (SNAPSHOT) 2.3.6 2.2.12 (SNAPSHOT) 2.2.11

Project Metadata

Group: com.thehecklers
Artifact: sbur-mongo
Name: sbur-mongo
Description: Kotlin and MongoDB demo project for Spring Boot
Package name: com.thehecklers.sbur-mongo
Packaging: [✓ Jar] War
Java: 15 [✓ 11] 8

Dependencies — ADD DEPENDENCIES... ⌘ + B

Spring Reactive Web WEB
Build reactive web applications with Spring WebFlux and Netty.

Spring Data MongoDB NOSQL
Store data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time.

Embedded MongoDB Database TESTING
Provides a platform neutral way for running MongoDB in unit tests.

Figure 6-1. Using the Spring Boot Initializr to create a Kotlin application

A couple of things of particular note about the options selected: First, I chose Gradle for this project's build system for good reason—simply choosing to use Gradle with Kotlin in a Spring Boot project results in the Gradle build file using the Kotlin DSL, which is supported by the Gradle team on equal footing with the Groovy DSL. Note that the resultant build file is *build.gradle.kts*—the .kts extension indicates it is a Kotlin script—rather than the Groovy-based *build.gradle* file you may be accustomed to seeing. Maven works perfectly well as a build system for Spring Boot + Kotlin applications too, but being an XML-based declarative build system, it doesn't directly use Kotlin or any other language.

Second, I took advantage of the presence of a Spring Boot Starter for an embedded MongoDB database for this application. Because an embedded MongoDB instance is meant solely for testing, I advise against using it in a production setting; that said, it's a wonderful option for demonstrating how Spring Boot and Spring Data work with MongoDB, and from the developer's perspective, it matches locally deployed database capabilities without the additional steps of installing and/or running a containerized instance of MongoDB. The only adjustment necessary to use the embedded database from (nontest) code is to change a single line in *build.gradle.kts* from this:

```
testImplementation("de.flapdoodle.embed:de.flapdoodle.embed.mongo")
```

to this:

```
implementation("de.flapdoodle.embed:de.flapdoodle.embed.mongo")
```

And with that, we're ready to create our service.

Developing the MongoDB Service

As with previous examples, the MongoDB-based service offers a very consistent approach and experience, even when using Kotlin instead of Java as the language foundation.

Defining the domain class

For this project I create a Kotlin `Aircraft` domain class to serve as the primary (data) focus. Here is the new `Aircraft` domain class structure with a few observations following:

```
import com.fasterxml.jackson.annotation.JsonIgnoreProperties
import com.fasterxml.jackson.annotation.JsonProperty
import org.springframework.data.annotation.Id
import org.springframework.data.mongodb.core.mapping.Document
import java.time.Instant

@Document
@JsonIgnoreProperties(ignoreUnknown = true)
data class Aircraft(
    @Id val id: String,
    val callsign: String? = "",
    val squawk: String? = "",
    val reg: String? = "",
    val flightno: String? = "",
    val route: String? = "",
    val type: String? = "",
    val category: String? = "",
    val altitude: Int? = 0,
    val heading: Int? = 0,
    val speed: Int? = 0,
    @JsonProperty("vert_rate") val vertRate: Int? = 0,
    @JsonProperty("selected_altitude")
    val selectedAltitude: Int? = 0,
    val lat: Double? = 0.0,
    val lon: Double? = 0.0,
    val barometer: Double? = 0.0,
    @JsonProperty("polar_distance")
    val polarDistance: Double? = 0.0,
    @JsonProperty("polar_bearing")
    val polarBearing: Double? = 0.0,
    @JsonProperty("is_adsb")
    val isADSB: Boolean? = false,
    @JsonProperty("is_on_ground")
    val isOnGround: Boolean? = false,
    @JsonProperty("last_seen_time")
    val lastSeenTime: Instant? = Instant.ofEpochSecond(0),
    @JsonProperty("pos_update_time")
    val posUpdateTime: Instant? = Instant.ofEpochSecond(0),
    @JsonProperty("bds40_seen_time")
    val bds40SeenTime: Instant? = Instant.ofEpochSecond(0)
)
```

The first thing to note is that there are no curly braces to be seen; put succinctly, this class has no body. If you're new to Kotlin, this may seem a bit unusual, but in cases where there is nothing to place in a class (or interface) body, curly braces add no value. As such, Kotlin doesn't require them.

The second interesting thing is the many assignments shown between parentheses immediately after the classname. What purpose do these serve?

A Kotlin class's primary constructor is often shown this way: in the class header, immediately following the classname. Here is an example of the full, formal format:

```
class Aircraft constructor(<parameter1>,<parameter2>,...,<parametern>)
```

As is often the case in Kotlin, if a pattern is clearly identifiable and repeats consistently, it can be condensed. Removing the `constructor` keyword before the parameter list leads to no confusion with any other language construct, so it is optional.

Within the constructor are parameters. By placing a `var` (for repeatedly assignable mutable variables) or `val` (for single-assignment values equivalent to Java's `final` variables) before each parameter, it also becomes a property. A Kotlin property is roughly equivalent in function to a Java member variable, its accessor, and (if declared with `var`) its mutator combined.

The values with types containing a question mark (?), e.g., `Double?`, indicate that the constructor parameter may be omitted. If so, that parameter is assigned the default value shown after the equals sign (=).

Kotlin method (including constructor) parameters and properties can also include annotations, just like their Java counterparts. `@Id` and

`@JsonProperty` perform the same functions that they did in earlier Java examples.

Regarding class-level annotations, `@Document` indicates to MongoDB that each object of type `Aircraft` will be stored as a document within the database. As before, `@JsonIgnoreProperties(ignoreUnknown = true)` simply builds a bit of flexibility into the *sbur-mongo* service; if at some point additional fields are added to the data feed produced by the upstream `PlaneFinder` service, they will simply be ignored and *sbur_-mongo* will continue to run without issue.

The final point of note is the word `data` that precedes the class definition. It's a frequent pattern to create domain classes that serve primarily as data buckets to be manipulated and/or passed between processes. It's such a common pattern in fact that the capability to create so-called data classes manifests itself in several ways; as one example, `@Data` has been a feature of Lombok for years.

Kotlin rolled this capability into the language itself and added the `data` keyword to signal that a data class automatically derives the following from all properties declared in the class's primary constructor:

- `equals()` and `hashCode()` functions (Java has methods; Kotlin has functions)
- `toString()`
- `componentN()` functions, one for each property in the order in which they were declared
- `copy()` function

Kotlin data classes have certain requirements and limitations, but they are reasonable and minimal. For details, please refer to the Kotlin documentation for data classes.

NOTE

One other change of interest is the type of each aircraft position's `id` field/property. In Redis and JPA, it was a `Long`; but MongoDB uses a `String` for its unique document identifier. This is of no real consequence, only something to be aware of.

Creating the repository interface

Next, I define the required repository interface, extending Spring Data's `CrudRepository` and providing the type of object to store and its unique identifier: `Aircraft` and `String`, as mentioned earlier:

```
interface AircraftRepository: CrudRepository<Aircraft, String>
```

There are two things of interest in this concise interface definition:

1. With no actual interface body, no curly braces are required in Kotlin. If your IDE added them when you created this interface, you can safely remove them.
2. Kotlin uses the colon (:) contextually to indicate a `val` or `var` type, or in this case, to indicate that a class or interface extends or implements another. In this particular instance, I define an interface `AircraftRepository`, and it extends the `CrudRepository` interface.

NOTE

There is a `MongoRepository` interface that extends both `PagingAndSortingRepository` (which extends `CrudRepository`) and `QueryByExampleExecutor` that can be used instead of `CrudRepository`, as I do here. But unless the additional capabilities are required, it is a good practice and habit to write to the highest-level interface that satisfies all requirements. In this case, `CrudRepository` is sufficient for current needs.

Bringing it all together

The next step is to create the component that periodically polls the `PlaneFinder` service.

Polling PlaneFinder

Similar to earlier examples, I create a Spring Boot component class `PlaneFinderPoller` to poll for current position data and handle any `Aircraft` records received, as shown here:

```
import org.springframework.scheduling.annotation.EnableScheduling
import org.springframework.scheduling.annotation.Scheduled
import org.springframework.stereotype.Component
import org.springframework.web.reactive.function.client.WebClient
import org.springframework.web.reactive.function.client.bodyToFlux

@Component
@EnableScheduling
class PlaneFinderPoller(private val repository: AircraftRepository) {
    private val client =
        WebClient.create("http://localhost:7634/aircraft")

    @Scheduled(fixedRate = 1000)
    private fun pollPlanes() {
        repository.deleteAll()

        client.get()
            .retrieve()
            .bodyToFlux<Aircraft>()
            .filter { !it.reg.isNullOrEmpty() }
            .toStream()
            .forEach { repository.save(it) }

        println("--- All aircraft ---")
        repository.findAll().forEach { println(it) }
    }
}
```

I create the primary constructor in the header with an `AircraftRepository` parameter. Spring Boot automatically autowires the existing `AircraftRepository` bean into the `PlaneFinderPoller` component for use, and I mark it as a `private val` to ensure the following:

- It isn't assignable later.
- It isn't exposed externally as a property from the `PlaneFinderPoller` bean, as the repository is already accessible throughout the application.

Next, I create a `WebClient` object and assign it to a property, pointing it to the destination endpoint exposed by the `PlaneFinder` service on port 7634.

I annotate the class with `@Component` to have Spring Boot create a component (bean) upon application startup and `@EnableScheduling` to enable periodic polling via an annotated function to follow.

And finally, I create a function to delete all existing `Aircraft` data, poll the `PlaneFinder` endpoint via the `WebClient` client property, convert and store the retrieved aircraft positions in MongoDB, and display them. The `@Scheduled(fixedRate = 1000)` results in the polling function being executed once every 1,000 ms (once per second).

There are three more interesting things to note in the `pollPlanes()` function, and both regard Kotlin's lambdas.

First is that if a lambda is the final parameter of a function, parentheses can be omitted, as they add nothing to clarity or meaning. If a function has only a single parameter of a lambda, this fits the criteria as well, of course. This results in fewer symbols to sift through in sometimes busy lines of code.

Second is that if a lambda itself has a single parameter, a developer can still explicitly specify it but isn't required to do so. Kotlin implicitly recognizes and refers to a sole lambda parameter as `it`, which further streamlines lambdas, as demonstrated by this lambda parameter to `forEach()`:

```
forEach { repository.save(it) }
```

Finally, the function `isNullOrEmpty()` that operates on a `CharSequence` provides a very nice all-in-one capability for String evaluation. This function performs both a null check (first), then if the value is determined to be non-null, it checks to see if it has zero length, i.e., is empty. There are many times that a developer can process properties only if they contain actual values, and this single function performs both validations in one step. If a value exists in the `Aircraft`'s registration property `reg`, that incoming aircraft position report is passed along; aircraft position reports with missing registration values are filtered out.

All remaining position reports are streamed to the repository to be saved, then we query the repository for all persisted documents and display the results.

The results

With the `PlaneFinder` service running on my machine, I start the *sbur-mongo* service to obtain, store and retrieve (in an embedded MongoDB instance), and display the results of each polling of `PlaneFinder`. An example of the results follows, edited for brevity and formatted for readability:

```
Aircraft(id=95, callsign=N88846, squawk=4710, reg=N88846, flightno=, route=
type=P46T, category=A1, altitude=18000, heading=234, speed=238, vertRate=-6
selectedAltitude=0, lat=39.157288, lon=-90.844992, barometer=0.0,
polarDistance=33.5716, polarBearing=290.454061, isADSB=true, isOnGround=fal
lastSeenTime=2020-11-27T20:16:57Z, posUpdateTime=2020-11-27T20:16:57Z,
bds40SeenTime=1970-01-01T00:00:00Z)

Aircraft(id=96, callsign=MVJ710, squawk=1750, reg=N710MV, flightno=,
route=IAD-TEX, type=GLF4, category=A2, altitude=18050, heading=66, speed=36
vertRate=2432, selectedAltitude=23008, lat=38.627655, lon=-90.008897,
barometer=0.0, polarDistance=20.976944, polarBearing=158.35465, isADSB=true
isOnGround=false, lastSeenTime=2020-11-27T20:16:57Z,
posUpdateTime=2020-11-27T20:16:57Z, bds40SeenTime=2020-11-27T20:16:56Z)
```

```
Aircraft(id=97, callsign=SWA1121, squawk=6225, reg=N8654B, flightno=WN1121,
route=MDW-DAL-PHX, type=B738, category=A3, altitude=40000, heading=236,
speed=398, vertRate=0, selectedAltitude=40000, lat=39.58548, lon=-90.049259
barometer=1013.6, polarDistance=38.411587, polarBearing=8.70042, isADSB=tru
isOnGround=false, lastSeenTime=2020-11-27T20:16:57Z,
posUpdateTime=2020-11-27T20:16:55Z, bds40SeenTime=2020-11-27T20:16:54Z)
```

As expected, the service polls, captures, and displays aircraft positions without issue using Spring Boot, Kotlin, and MongoDB to make it nearly effortless.

Creating a Repository-Based Service Using a NoSQL Graph Database

Graph databases bring a different approach to data, in particular how it's interrelated. There are a few graph databases on the market, but for all intents and purposes, the segment leader is Neo4j.

While graph theory and graph database design is far afield of the scope of this book, demonstrating how best to work with a graph database using Spring Boot and Spring Data falls squarely within its purview. This section shows you how to easily connect to and work with data using Spring Data Neo4j in your Spring Boot application.

Initializing the Project

Once more we return to the Spring Initializr. This time, I choose the following options:

- Gradle project
- Java
- Current production version of Spring Boot
- Packaging: Jar
- Java: 11

And for dependencies:

- Spring Reactive Web (`spring-boot-starter-webflux`)
- Spring Data Neo4j (`spring-boot-starter-data-neo4j`)

Next, I generate the project and save it locally, unzip it, and open it in the IDE.

I chose Gradle for this project's build system solely to demonstrate that when creating a Spring Boot Java application using Gradle, the generated *build.gradle* file uses the Groovy DSL, but Maven is a valid option as well.

NOTE

As with most other examples in this chapter, I have a Neo4j database instance running in a locally hosted container, ready to respond to this application.

And with that, we're ready to create our service.

Developing the Neo4j Service

As with previous examples, Spring Boot and Spring Data make the experience of working with Neo4j databases highly consistent with using other types of underlying datastores. The full power of a graph datastore is available and easily accessible from Spring Boot applications, but ramp-up is drastically reduced.

Defining the domain class

Once more I begin by defining the `Aircraft` domain. Without Lombok as a dependency, I create it with the usual extensive list of constructors, accessors, mutators, and supporting methods:

```java
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
import org.springframework.data.neo4j.core.schema.GeneratedValue;
import org.springframework.data.neo4j.core.schema.Id;
import org.springframework.data.neo4j.core.schema.Node;

@Node
@JsonIgnoreProperties(ignoreUnknown = true)
public class Aircraft {
    @Id
    @GeneratedValue
    private Long neoId;

    private Long id;
    private String callsign, squawk, reg, flightno, route, type, category;

    private int altitude, heading, speed;
    @JsonProperty("vert_rate")
    private int vertRate;
    @JsonProperty("selected_altitude")
    private int selectedAltitude;
    private double lat, lon, barometer;
    @JsonProperty("polar_distance")
    private double polarDistance;
    @JsonProperty("polar_bearing")
    private double polarBearing;

    @JsonProperty("is_adsb")
    private boolean isADSB;
    @JsonProperty("is_on_ground")
    private boolean isOnGround;

    @JsonProperty("last_seen_time")
    private Instant lastSeenTime;
    @JsonProperty("pos_update_time")
    private Instant posUpdateTime;
    @JsonProperty("bds40_seen_time")
    private Instant bds40SeenTime;

    public Aircraft() {
    }

    public Aircraft(Long id,
                    String callsign, String squawk, String reg, String flig
                    String route, String type, String category,
                    int altitude, int heading, int speed,
                    int vertRate, int selectedAltitude,
                    double lat, double lon, double barometer,
                    double polarDistance, double polarBearing,
                    boolean isADSB, boolean isOnGround,
                    Instant lastSeenTime,
                    Instant posUpdateTime,
                    Instant bds40SeenTime) {
        this.id = id;
        this.callsign = callsign;
        this.squawk = squawk;
        this.reg = reg;
        this.flightno = flightno;
        this.route = route;
        this.type = type;
        this.category = category;
        this.altitude = altitude;
        this.heading = heading;
        this.speed = speed;
        this.vertRate = vertRate;
        this.selectedAltitude = selectedAltitude;
        this.lat = lat;
        this.lon = lon;
        this.barometer = barometer;
        this.polarDistance = polarDistance;
        this.polarBearing = polarBearing;
        this.isADSB = isADSB;
        this.isOnGround = isOnGround;
        this.lastSeenTime = lastSeenTime;
        this.posUpdateTime = posUpdateTime;
        this.bds40SeenTime = bds40SeenTime;
    }

    public Long getNeoId() {
        return neoId;
    }

    public void setNeoId(Long neoId) {
        this.neoId = neoId;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getCallsign() {
        return callsign;
    }

    public void setCallsign(String callsign) {
        this.callsign = callsign;
    }

    public String getSquawk() {
        return squawk;
    }

    public void setSquawk(String squawk) {
        this.squawk = squawk;
    }

    public String getReg() {
        return reg;
    }

    public void setReg(String reg) {
        this.reg = reg;
    }

    public String getFlightno() {
        return flightno;
    }

    public void setFlightno(String flightno) {
        this.flightno = flightno;
    }
    public String getRoute() {
        return route;
    }

    public void setRoute(String route) {
        this.route = route;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public int getAltitude() {
        return altitude;
    }

    public void setAltitude(int altitude) {
        this.altitude = altitude;
    }

    public int getHeading() {
        return heading;
    }

    public void setHeading(int heading) {
        this.heading = heading;
    }

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int getVertRate() {
        return vertRate;
    }
```

```java
    public void setVertRate(int vertRate) {
        this.vertRate = vertRate;
    }

    public int getSelectedAltitude() {
        return selectedAltitude;
    }

    public void setSelectedAltitude(int selectedAltitude) {
        this.selectedAltitude = selectedAltitude;
    }

    public double getLat() {
        return lat;
    }

    public void setLat(double lat) {
        this.lat = lat;
    }

    public double getLon() {
        return lon;
    }

    public void setLon(double lon) {
        this.lon = lon;
    }

    public double getBarometer() {
        return barometer;
    }

    public void setBarometer(double barometer) {
        this.barometer = barometer;
    }

    public double getPolarDistance() {
        return polarDistance;
    }

    public void setPolarDistance(double polarDistance) {
        this.polarDistance = polarDistance;
    }

    public double getPolarBearing() {
        return polarBearing;
    }

    public void setPolarBearing(double polarBearing) {
        this.polarBearing = polarBearing;
    }
    public boolean isADSB() {
        return isADSB;
    }

    public void setADSB(boolean ADSB) {
        isADSB = ADSB;
    }

    public boolean isOnGround() {
        return isOnGround;
    }

    public void setOnGround(boolean onGround) {
        isOnGround = onGround;
    }

    public Instant getLastSeenTime() {
        return lastSeenTime;
    }

    public void setLastSeenTime(Instant lastSeenTime) {
        this.lastSeenTime = lastSeenTime;
    }

    public Instant getPosUpdateTime() {
        return posUpdateTime;
    }

    public void setPosUpdateTime(Instant posUpdateTime) {
        this.posUpdateTime = posUpdateTime;
    }

    public Instant getBds40SeenTime() {
        return bds40SeenTime;
    }

    public void setBds40SeenTime(Instant bds40SeenTime) {
        this.bds40SeenTime = bds40SeenTime;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Aircraft aircraft = (Aircraft) o;
        return altitude == aircraft.altitude &&
                heading == aircraft.heading &&
                speed == aircraft.speed &&
                vertRate == aircraft.vertRate &&
                selectedAltitude == aircraft.selectedAltitude &&
                Double.compare(aircraft.lat, lat) == 0 &&
                Double.compare(aircraft.lon, lon) == 0 &&
                Double.compare(aircraft.barometer, barometer) == 0 &&
                Double.compare(aircraft.polarDistance, polarDistance) == 0
                Double.compare(aircraft.polarBearing, polarBearing) == 0 &&
                isADSB == aircraft.isADSB &&
                isOnGround == aircraft.isOnGround &&
                Objects.equals(neoId, aircraft.neoId) &&
                Objects.equals(id, aircraft.id) &&
                Objects.equals(callsign, aircraft.callsign) &&
                Objects.equals(squawk, aircraft.squawk) &&
                Objects.equals(reg, aircraft.reg) &&
                Objects.equals(flightno, aircraft.flightno) &&
                Objects.equals(route, aircraft.route) &&
                Objects.equals(type, aircraft.type) &&
                Objects.equals(category, aircraft.category) &&
                Objects.equals(lastSeenTime, aircraft.lastSeenTime) &&
                Objects.equals(posUpdateTime, aircraft.posUpdateTime) &&
                Objects.equals(bds40SeenTime, aircraft.bds40SeenTime);
    }

    @Override
    public int hashCode() {
        return Objects.hash(neoId, id, callsign, squawk, reg, flightno, rou
                type, category, altitude, heading, speed, vertRate,
                selectedAltitude,  lat, lon, barometer, polarDistance,
                polarBearing, isADSB, isOnGround, lastSeenTime, posUpdateTi
                bds40SeenTime);
    }

    @Override
    public String toString() {
        return "Aircraft{" +
                "neoId=" + neoId +
                ", id=" + id +
                ", callsign='" + callsign + '\'' +
                ", squawk='" + squawk + '\'' +
                ", reg='" + reg + '\'' +
                ", flightno='" + flightno + '\'' +
                ", route='" + route + '\'' +
                ", type='" + type + '\'' +
                ", category='" + category + '\'' +
                ", altitude=" + altitude +
                ", heading=" + heading +
                ", speed=" + speed +
                ", vertRate=" + vertRate +
                ", selectedAltitude=" + selectedAltitude +
                ", lat=" + lat +
                ", lon=" + lon +
                ", barometer=" + barometer +
                ", polarDistance=" + polarDistance +
                ", polarBearing=" + polarBearing +
                ", isADSB=" + isADSB +
                ", isOnGround=" + isOnGround +
                ", lastSeenTime=" + lastSeenTime +
                ", posUpdateTime=" + posUpdateTime +
                ", bds40SeenTime=" + bds40SeenTime +
                '}';
    }
}
```

Java code can indeed be verbose. To be fair this isn't a huge problem in cases like domain classes, because while accessors and mutators take up a significant amount of space, they can be generated by IDEs and typically don't involve much maintenance due to their long-term stability. That said, it *is* a lot of boilerplate code, which is why many developers use solutions like Lombok or Kotlin—even if only creating domain classes in Kotlin for Java applications.

NOTE

Neo requires a database-generated unique identifier, even if entities being persisted contain a unique identifier already. To satisfy this requirement, I add a `neoId` parameter/member variable and annotate it with `@Id` and `GeneratedValue` so Neo4j correctly associates this member variable with the value it generates internally.

Next, I add two class-level annotations:

`@Node` :: To designate each instance of this `record` as an instance of the Neo4j node `Aircraft` `@JsonIgnoreProperties(ignoreUnknown = true)` :: To ignore new fields that might be added to feed from the `PlaneFinder` service endpoint

Note that like `@Id` and `@GeneratedValue`, the `@Node` annotation is from the `org.springframework.data.neo4j.core.schema` package for Spring Data Neo4j-based applications.

With that, the domain for our service is defined.

Creating the repository interface

For this application I again define the required repository interface, extending Spring Data's `CrudRepository` and providing the type of object to store and its key: `Aircraft` and `Long`, in this case:

```
public interface AircraftRepository extends CrudRepository<Aircraft, Long>
```

NOTE

Similar to the earlier MongoDB-based project, there is a `Neo4jRepository` interface that extends `PagingAndSortingRepository` (which extends `CrudRepository`) that can used instead of `CrudRepository`; however, since `CrudRepository` is the highest-level interface that satisfies all requirements, I use it as the basis for `AircraftRepository`.

Bringing it all together

Now to create the component to poll `PlaneFinder` and configure it to access the Neo4j database.

Polling PlaneFinder

Once more I create a Spring Boot `@Component` class to poll for current aircraft positions and handle `Aircraft` records received.

Like other Java-based projects in this chapter, I create a `WebClient` object and assign it to a member variable, pointing it to the destination endpoint exposed by the `PlaneFinder` service on port 7634.

Without Lombok as a dependency, I create a constructor via which to receive the autowired `AircraftRepository` bean.

As shown in the following full listing of the `PlaneFinderPoller` class, the `pollPlanes()` method looks nearly identical to other examples, owing to the abstractions brought to bear by repository support. To revisit any other details of the remaining code in `PlaneFinderPoller`, please review the corresponding section under earlier sections:

```
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@EnableScheduling
@Component
public class PlaneFinderPoller {
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");
    private final AircraftRepository repository;

    public PlaneFinderPoller(AircraftRepository repository) {
        this.repository = repository;
    }

    @Scheduled(fixedRate = 1000)
    private void pollPlanes() {
        repository.deleteAll();

        client.get()
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(plane -> !plane.getReg().isEmpty())
                .toStream()
                .forEach(repository::save);

        System.out.println("--- All aircraft ---");
        repository.findAll().forEach(System.out::println);
    }
}
```

Connecting to Neo4j

As with the earlier MariaDB/MySQL example, we need to provide a few key bits of information for Boot to use to seamlessly connect to a Neo4j database. For this service running in my environment, these properties include:

```
spring.neo4j.authentication.username=neo4j
spring.neo4j.authentication.password=mkheck
```

NOTE

Replace username and password values shown with those specific to your environment.

The results

With the `PlaneFinder` service running on my machine, I start the *sbur-neo* service to obtain, store and retrieve, and display the results of each polling of `PlaneFinder` using Neo4j as the datastore of choice. An example of the results follows, edited for brevity and formatted for readability:

```
Aircraft(neoId=64, id=223, callsign='GJS4401', squawk='1355', reg='N542GJ',
flightno='UA4401', route='LIT-ORD', type='CRJ7', category='A2', altitude=37
heading=24, speed=476, vertRate=128, selectedAltitude=36992, lat=39.463961,
lon=-90.549927, barometer=1012.8, polarDistance=35.299257,
polarBearing=329.354686, isADSB=true, isOnGround=false,
lastSeenTime=2020-11-27T20:42:54Z, posUpdateTime=2020-11-27T20:42:53Z,
bds40SeenTime=2020-11-27T20:42:51Z)

Aircraft(neoId=65, id=224, callsign='N8680B', squawk='1200', reg='N8680B',
flightno='', route='', type='C172', category='A1', altitude=3100, heading=1
speed=97, vertRate=64, selectedAltitude=0, lat=38.923955, lon=-90.195618,
barometer=0.0, polarDistance=1.986086, polarBearing=208.977102, isADSB=true
isOnGround=false, lastSeenTime=2020-11-27T20:42:54Z,
posUpdateTime=2020-11-27T20:42:54Z, bds40SeenTime=null)

Aircraft(neoId=66, id=225, callsign='AAL1087', squawk='1712', reg='N181UW',
flightno='AA1087', route='CLT-STL-CLT', type='A321', category='A3',
altitude=7850, heading=278, speed=278, vertRate=-320, selectedAltitude=4992
lat=38.801559, lon=-90.226474, barometer=0.0, polarDistance=9.385111,
polarBearing=194.034005, isADSB=true, isOnGround=false,
lastSeenTime=2020-11-27T20:42:54Z, posUpdateTime=2020-11-27T20:42:53Z,
bds40SeenTime=2020-11-27T20:42:53Z)
```

The service is fast and efficient, using Spring Boot and Neo4j to retrieve, capture, and display aircraft positions as they're reported.

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter6end* from the code repository.

Summary

Data can be a complex topic with innumerable variables and constraints, including data structures, relationships, applicable standards, providers and mechanisms, and more. Yet without data in some form, most applications provide little or no value.

As something that forms the foundation of nearly all application value, "data" has attracted a great deal of innovation from database providers and platform vendors. In many cases, though, complexity remains, and developers have to tame that complexity to unlock the value.

Spring Data's stated mission is "to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store." Regardless of database engine or platform, Spring Data's goal is to make the developer's use of data as simple and as powerful as humanly possible.

This chapter demonstrated how to streamline data storage and retrieval using various database options and the Spring Data projects and facilities that enable their use in the most powerful ways possible: via Spring Boot.

In the next chapter, I’ll show how to create imperative applications using Spring MVC’s REST interactions, messaging platforms, and other communications mechanisms, as well as provide an introduction to templating language support. While this chapter’s focus was from the application downward, Chapter 7 focuses on the application outward.