Chapter 7. Creating Applications Using Spring MVC

This chapter demonstrates how to create Spring Boot applications using Spring MVC with REST interactions, messaging platforms, and other communications mechanisms and provides an introduction to templating language support. Although I introduced interservice interactions as part of last chapter's dive into Spring Boot's many options for handling data, this chapter shifts the primary focus from the application itself to the outside world: its interactions with other applications and/or services and with end users.

CODE CHECKOUT CHECKUP

Please check out branch *chapter7begin* from the code repository to begin.

Spring MVC: What Does It Mean?

Like many other things in technology, the term *Spring MVC* is somewhat overloaded. When someone refers to Spring MVC, they could mean any of the following:

- Implementing (in some manner) the Model-View-Controller pattern in a Spring application
- Creating an application specifically using Spring MVC component concepts like the `Model` interface, `@Controller` classes, and view technologies
- Developing blocking/nonreactive applications using Spring

Depending on context, Spring MVC can be considered both an approach and an implementation. It can also be used within or without Spring Boot. Generic application of the MVC pattern using Spring and Spring MVC use outside of Spring Boot both fall outside the scope of this book. I'll focus specifically on the final two concepts previously listed using Spring Boot to implement them.

End User Interactions Using Template

Engines

While Spring Boot applications handle a lot of heavy-lifting chores on the backend, Boot also supports direct end-user interactions as well. Although long-established standards like Java Server Pages (JSP) are still supported by Boot for legacy applications, most current applications either leverage more powerful view technologies supported by still-evolving and -maintained template engines or shift frontend development to a combination of HTML and JavaScript. It's even possible to mix the two options successfully and play to each one's strengths.

Spring Boot works well with HTML and JavaScript frontends, as I demonstrate later in this chapter. For now, let's take a closer look at template engines.

Template engines provide a way for a so-called server-side application to generate the final pages that will be displayed and executed in the end user's browser. These view technologies differ in approaches but generally provide the following:

- A template language and/or collection of tags that define inputs used by the template engine to produce the expected outcome
- A view resolver that determines the view/template to use to fulfill a requested resource

Among other lesser-used options, Spring Boot supports view technologies such as Thymeleaf, FreeMarker, Groovy Markup, and Mustache. Thymeleaf is perhaps the most widely used of these for several reasons and provides excellent support for both Spring MVC and Spring WebFlux applications.

Thymeleaf uses natural templates: files that incorporate code elements but that can be opened and viewed directly (and correctly) in any standard web browser. Being able to view the template files as HTML enables developers or designers to create and evolve Thymeleaf templates without any running server processes. Any code integrations that expect corresponding server-side elements are tagged as Thymeleaf-specific and simply don't display what isn't present.

Building on previous efforts, let's build a simple web application using Spring Boot, Spring MVC, and Thymeleaf to present to the end user an interface for querying PlaneFinder for current aircraft positions and displaying the results. Initially this will be a rudimentary proof of concept to be evolved in subsequent chapters.

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 Web (`spring-boot-starter-web`)
- Spring Reactive Web (`spring-boot-starter-webflux`)
- Thymeleaf (`spring-boot-starter-thymeleaf`)
- Spring Data JPA (`spring-boot-starter-data-jpa`)
- H2 Database (`h2`)
- Lombok (`lombok`)

The next step is to generate the project and save it locally, unzip it, and open it in the IDE.

Developing the Aircraft Positions Application

Since this application is concerned only with the current state—aircraft positions at the moment the request is made, not historically—an in-memory database seems a reasonable choice. One could instead use an `Iterable` of some kind, of course, but Spring Boot's support for Spring Data repositories and the H2 database fulfill the current use case and position the application well for planned future expansion.

Defining the domain class

As with other projects interacting with `PlaneFinder`, I create an `Aircraft` domain class to serve as the primary (data) focus. Here is the `Aircraft` domain class structure for the `Aircraft Positions` application:

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
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;
}
```

This domain class is defined using JPA with H2 as the underlying JPA-compliant database and leveraging Lombok to create a data class with constructors having zero arguments and all arguments, one for every member variable.

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>
```

Working with Model and Controller

I've defined the data behind the model with the `Aircraft` domain class; now it's time to incorporate it into the `Model` and expose it via a `Controller`.

As discussed in Chapter 3, `@RestController` is a convenience notation that combines `@Controller` with `@ResponseBody` into a single descriptive annotation, returning a formatted response as JavaScript Object Notation (JSON) or as other data-oriented format. This results in the Object/Iterable return value of a method being the *entire body* of the response to a web request, instead of being returned as a part of the `Model`. An `@RestController` enables the creation of an API, a specialized, but very common, use case.

The goal now is to create an application that also includes a user interface, and `@Controller` enables that. Within an `@Controller` class, each method annotated with `@RequestMapping` or one of its specialized aliases like `@GetMapping` will return a `String` value that corresponds to the name of a template file minus its extension. For example, Thymeleaf files have the *.html* file extension, so if an `@Controller` class's `@GetMapping` method returns the `String` "myfavoritepage", the Thymeleaf template engine will use the *myfavoritepage.html* template to create and return the generated page to the user's browser.

NOTE

View technology templates are placed under the project's *src/main/resources/templates* directory by default; the template engine will look here for them unless overridden via application properties or programmatic means.

Returning to the controller, I create a class `PositionController` as follows:

```java
@RequiredArgsConstructor
@Controller
public class PositionController {
    @NonNull
    private final AircraftRepository repository;
    private WebClient client =
            WebClient.create("http://localhost:7634/aircraft");

    @GetMapping("/aircraft")
    public String getCurrentAircraftPositions(Model model) {
        repository.deleteAll();

        client.get()
                .retrieve()
                .bodyToFlux(Aircraft.class)
                .filter(plane -> !plane.getReg().isEmpty())
                .toStream()
                .forEach(repository::save);
        model.addAttribute("currentPositions", repository.findAll());
        return "positions";
    }
}
```

This controller looks very similar to previous iterations but with a few key differences. First, of course, is the `@Controller` annotation previously discussed instead of `@RestController`. Second is that the `getCurrentAircraftPositions()` method has an automatically autowired parameter: `Model model`. This parameter is the `Model` bean that is leveraged by the template engine to provide access to the application's components—their data and operations—once we add those components to the `Model` as an attribute. And third is the method's return type of `String` instead of a class type and the actual return statement with the name of a template (sans *.html* extension).

NOTE

In a complex domain/application, I prefer to separate concerns a bit more by creating distinct `@Service` and `@Controller` classes. In this example, there is a single method making a single repository access, so I've placed all functionality to populate the underlying data, populate the `Model`, and hand it off to the appropriate `View` within the `Controller`.

Creating the requisite View files

As a basic foundation for this and future chapters, I create one plain HTML file and one template file.

Since I want to display a plain HTML page to all visitors, and since this page requires no template support, I place *index.html* directly in the project's *src/main/resources/static* directory:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Retrieve Aircraft Position Report</title>
</head>
<body>
    <p><a href="/aircraft">Click here</a>
        to retrieve current aircraft positions in range of receiver.</p>
</body>
</html>
```

NOTES ABOUT INDEX.HTML

By default, a Spring Boot application will look for static pages in the classpath under *static* and *public* directories. To properly place them there during build, place them within one of those two directories under *src/main/resources* within the project.

Of particular interest to this application is the `href` hyperlink "/aircraft". This link matches the `@GetMapping` annotation for the `PositionController` `getCurrentAircraftPositions()` method and points to the endpoint exposed by it, another example of the internal integration by Spring Boot across various components within the application. Clicking *Click here* from the page displayed by the running application will execute `getCurrentAircraftPositions()`, which will return "positions", prompting the `ViewResolver` to generate and return the next page based on the template *positions.html*.

As a final note, if an *index.html* file is located in one of the searched classpath directories, Spring Boot will automatically load it for the user when the application's *host:port* address is accessed from a browser or other user agent with no configuration required from the developer.

For the dynamic content, I create a template file, adding an XML namespace for Thymeleaf tags to the otherwise plain HTML file and then using those tags as content injection guidance for the Thymeleaf template engine, as shown in the following *positions.html* file. To designate this as a template file for processing by the engine, I place it in the *src/main/resources/templates* project directory:

```
<!DOCTYPE HTML>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Position Report</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
</head>
<body>
<div class="positionlist" th:unless="${#lists.isEmpty(currentPositions)}">

    <h2>Current Aircraft Positions</h2>

    <table>
        <thead>
        <tr>
            <th>Call Sign</th>
            <th>Squawk</th>
            <th>AC Reg</th>
```

```
                <th>Flight #</th>
                <th>Route</th>
                <th>AC Type</th>
                <th>Altitude</th>
                <th>Heading</th>
                <th>Speed</th>
                <th>Vert Rate</th>
                <th>Latitude</th>
                <th>Longitude</th>
                <th>Last Seen</th>
                <th></th>
            </tr>
            </thead>
            <tbody>
            <tr th:each="ac : ${currentPositions}">
                <td th:text="${ac.callsign}"></td>
                <td th:text="${ac.squawk}"></td>
                <td th:text="${ac.reg}"></td>
                <td th:text="${ac.flightno}"></td>
                <td th:text="${ac.route}"></td>
                <td th:text="${ac.type}"></td>
                <td th:text="${ac.altitude}"></td>
                <td th:text="${ac.heading}"></td>
                <td th:text="${ac.speed}"></td>
                <td th:text="${ac.vertRate}"></td>
                <td th:text="${ac.lat}"></td>
                <td th:text="${ac.lon}"></td>
                <td th:text="${ac.lastSeenTime}"></td>
            </tr>
            </tbody>
        </table>
</div>
</body>
</html>
```

For the aircraft position report page, I reduce the information displayed to a select few elements of particular importance and interest. There are a few items of note in the *positions.html* Thymeleaf template:

First, as mentioned earlier, I add the Thymeleaf tags to the XML namespace with the *th* prefix with the following line:

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

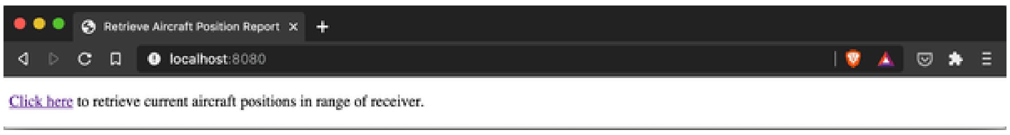
When defining the `division` that will display the current aircraft positions, I direct that the positionList division should be shown only if data is present; if the `currentPositions` element within the `Model` is empty, simply omit the entire division:

```
<div class="positionlist" th:unless="${#lists.isEmpty(currentPositions)}">
```

Finally, I define a table using standard HTML table tags for the table itself and the header row and its contents. For the table body, I use Thymeleaf's `each` to iterate through all `currentPositions` and populate each row's columns using the Thymeleaf's `text` tag and referencing each position object's properties via the "${object.property}" variable expression syntax. With that, the application is ready for testing.

The results

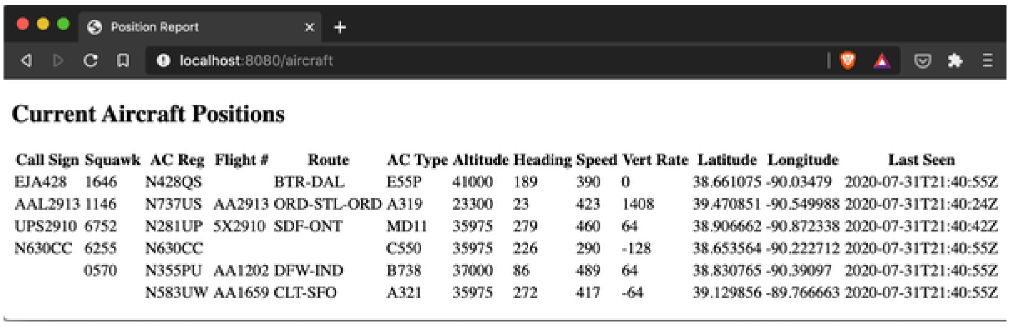
With the `PlaneFinder` service running, I execute the `Aircraft Positions` application from the IDE. Once it has successfully started, I open a browser tab and enter `localhost:8080` in the address bar and hit enter. Figure 7-1 shows the resultant page.



Click here to retrieve current aircraft positions in range of receiver.

Figure 7-1. The Aircraft Positions application (very simple) landing page

From here, I click the *Click here* link to proceed to the Aircraft Position Report page, as shown in Figure 7-1.



Current Aircraft Positions

Call Sign	Squawk	AC Reg	Flight #	Route	AC Type	Altitude	Heading	Speed	Vert Rate	Latitude	Longitude	Last Seen
EJA428	1646	N428QS		BTR-DAL	E55P	41000	189	390	0	38.661075	-90.03479	2020-07-31T21:40:55Z
AAL2913	1146	N737US	AA2913	ORD-STL-ORD	A319	23300	23	423	1408	39.470851	-90.549988	2020-07-31T21:40:24Z
UPS2910	6752	N281UP	5X2910	SDF-ONT	MD11	35975	279	460	64	38.906662	-90.872338	2020-07-31T21:40:42Z
N630CC	6255	N630CC			C550	35975	226	290	-128	38.653564	-90.222712	2020-07-31T21:40:55Z
	0570	N355PU	AA1202	DFW-IND	B738	37000	86	489	64	38.830765	-90.39097	2020-07-31T21:40:55Z
		N583UW	AA1659	CLT-SFO	A321	35975	272	417	-64	39.129856	-89.766663	2020-07-31T21:40:55Z

Figure 7-2. The Aircraft Position Report Page

Refreshing the page will requery `PlaneFinder` and update the report with current data on demand.

A refreshing flourish

Being able to request a listing of aircraft currently in the area along with their exact positions is a useful thing. But having to manually refresh the page could also become quite tedious and result in missing data of great interest, if one is so disposed. To add a timed refresh function to the Aircraft Position Report template, simply add a JavaScript function to the page `body` similar to the following, specifying the page refresh rate in milliseconds:

```
<script type="text/javascript">
    window.onload = setupRefresh;

    function setupRefresh() {
        setTimeout("refreshPage();", 5000); // refresh rate in milliseconds
    }

    function refreshPage() {
        window.location = location.href;
    }
</script>
```

The Thymeleaf template engine passes this code into the generated page untouched, and the user's browser executes the script at the designated refresh rate. It isn't the most elegant solution, but for simple use cases, it does the job.

Passing Messages

When use cases are a bit more demanding, more sophisticated solutions may be required. The preceding code does provide dynamic updates reflecting the latest available position data, but among other potential concerns are that periodic requests for updated data can be somewhat chatty. If several clients are requesting and receiving updates constantly, network traffic can be substantial.

In order to fulfill more complex use cases while simultaneously addressing network demands, it's helpful to shift perspectives: from a pull model to a push model, or some combination of the two.

NOTE

This section and the next explore two different and incremental steps toward a push model, culminating in an *entirely* push-based model from the `PlaneFinder` service outward. Use cases will indicate (or dictate) conditions that may favor one of these approaches or something else entirely. I continue to explore and demonstrate additional alternatives in subsequent chapters, so stay tuned.

Messaging platforms were made to efficiently accept, route, and deliver messages between applications. Examples include RabbitMQ and Apache Kafka and numerous other offerings, both open source and commercial.

Spring Boot and the Spring ecosystem provide a few different options for leveraging message pipelines, but my hands-down favorite is Spring Cloud Stream.

Spring Cloud Stream elevates the level of abstraction for developers while still providing access to supported platforms' unique attributes via application properties, beans, and direct configuration. Binders form the connection between streaming platform drivers and Spring Cloud Stream (SCSt), allowing developers to maintain focus on the key tasks—sending, routing, and receiving messages—which don't differ in concept regardless of the underlying plumbing.

Powering Up PlaneFinder

The first order of business is to refactor the `PlaneFinder` service to use Spring Cloud Stream to publish messages for consumption by the `Aircraft Positions` (and any other applicable) application.

Required dependencies

I add the following dependencies to `PlaneFinder` 's *pom.xml* Maven build file:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

The first thing to note is actually the second dependency listed: `spring-cloud-stream`. This is the code dependency for Spring Cloud Stream, but it can't do the job alone. As mentioned, SCSt uses binders to enable its powerful abstraction to work with various streaming platforms' drivers seamlessly. There is even a helpful reminder on the Spring Cloud Stream entry accessible from the Spring Initializr to that effect:

> *Framework for building highly scalable event-driven microservices connected with shared messaging systems (requires a binder, e.g., Apache Kafka, RabbitMQ, or Solace PubSub+)*

For Spring Cloud Stream to work with a messaging platform, it requires a messaging platform driver and the binder that works with it. In the preceding example, I include a binder+driver combination for RabbitMQ *and* for Apache Kafka.

TIP

If only one binder+driver combination is included—for RabbitMQ, for example—Spring Boot's autoconfiguration can unambiguously determine that your application should support communication with RabbitMQ instance(s) and associated `exchanges` and `queues` and create the appropriate supporting beans with no additional effort required on the developer's part. Including more than one set of binders+drivers requires us to specify which one to use, but it also allows us to dynamically switch among all included platforms at runtime, with no change to the tested and deployed application. This is an extremely powerful and useful capability.

Two more additions to the *pom.xml* file are necessary. First is to indicate the project-level version of Spring Cloud to use by adding this line to the `<properties></properties>` section:

```
<spring-cloud.version>2020.0.0-M5</spring-cloud.version>
```

Second is to provide guidance on the Spring Cloud Bill of Materials (BOM), from which the build system can determine versions for any Spring Cloud components—in this case, Spring Cloud Stream—that are used in this project:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

NOTE

Versions of Spring component projects are updated frequently. An easy way to determine correct, synchronized versions tested with the current version of Spring Boot is to use the Spring Initializr. Selecting the desired dependencies and clicking the button to "*Explore CTRL+SPACE*" displays the build file with the appropriate elements and versions.

After refreshing the project's dependencies, it's on to the code.

Supplying aircraft positions

Due to `PlaneFinder` 's existing structure and Spring Cloud Stream's clean, functional approach, only one small class is required to publish current aircraft positions to RabbitMQ for consumption by other applications:

```
@AllArgsConstructor
@Configuration
public class PositionReporter {
    private final PlaneFinderService pfService;

    @Bean
    Supplier<Iterable<Aircraft>> reportPositions() {
        return () -> {
            try {
                return pfService.getAircraft();
            } catch (IOException e) {
                e.printStackTrace();
            }
            return List.of();
        };
    }
}
```

A FEW THOUGHTS ON APPLICATION DESIGN

First, technically speaking, only the `reportPositions()` bean creation method is required, not the entire `PositionReporter` class. Since the main application class is annotated with `@SpringBootApplication`, a meta-annotation that incorporates `@Configuration` within, one could simply place `reportPositions()` within the main application class, `PlanefinderApplication`. My preference is to place `@Bean` methods within relevant `@Configuration` classes, especially in cases where numerous beans are created.

Second, Spring Cloud Stream's annotation-driven legacy API is still fully supported, but in this book I focus exclusively on the newer functional API. Spring Cloud Stream builds on the clean lines of Spring Cloud Function, which builds on *standard Java concepts/interfaces*: `Supplier<T>`, `Function<T, R>`, and `Consumer<T>`. This removes from SCSt the slightly leaky abstraction of Spring Integration concepts and supplants it with core language constructs; it also enables some new capabilities, as you might imagine.

Briefly stated, applications can either supply messages (`Supplier<T>`), transform messages (`Function<T, R>`) from one kind of thing to another, or consume messages (`Consumer<T>`). Any supported streaming platform can supply the connecting pipelines.

Platforms currently supported by Spring Cloud Stream include the following:

- RabbitMQ
- Apache Kafka
- Kafka Streams
- Amazon Kinesis
- Google Pub/Sub (partner maintained)
- Solace PubSub+ (partner maintained)
- Azure Event Hubs (partner maintained)
- Apache RocketMQ (partner maintained)

Since each poll by `PlaneFinder` of the upstream radio device produces a listing of positions of aircraft currently within range, the `PlaneFinder` service creates a message consisting of 1+ aircraft in an `Iterable<Aircraft>` by calling the `PlaneFinderService` `getAircraft()` method. An opinion—that a `Supplier` is called once per second by default (overridable via application property)—and some required/optional application properties inform Spring Boot's autoconfig-uration and set things in motion.

Application properties

Only one property is required, although others are helpful. Here are the contents of the updated `PlaneFinder` 's *application.properties* file:

```
server.port=7634

spring.cloud.stream.bindings.reportPositions-out-0.destination=aircraftposi
spring.cloud.stream.bindings.reportPositions-out-0.binder=rabbit
```

The `server.port` remains from the first version and indicates the application should listen on port 7634.

Spring Cloud Stream's functional API relies on minimal property configuration when necessary (as a baseline) to enable its functionality. A `Supplier` has only output channels, as it produces only messages. A `Consumer` has only input channels, as it consumes only messages. A `Function` has both input and output channels, which are necessary due to its use in transforming one thing to another.

Each binding uses the interface (`Supplier`, `Function`, or `Consumer`) bean method's name for the channel name, along with `in` or `out` and a channel number from `0` to `7`. Once concatenated in the form `<method>-<in|out>-n`, binding properties can be defined for the channel.

The only property required for this use case is `destination`, and even that is for convenience. Specifying the `destination` name results in RabbitMQ creating an exchange named `aircraftpositions` (in this example).

Since I included binders and drivers for both RabbitMQ and Kafka in the project dependencies, I must specify which binder the application should use. For this example, I choose `rabbit`.

With all required and desired application properties defined, `PlaneFinder` is ready to publish current aircraft positions each second to RabbitMQ for consumption by any applications desiring to do so.

Extending the Aircraft Positions Application

Converting `Aircraft Positions` to consume messages from a RabbitMQ pipeline using Spring Cloud Stream is similarly straightforward. Only a few changes to the workings behind the scenes are necessary to replace frequent HTTP requests with a message-driven architecture.

Required dependencies

Just as with `PlaneFinder`, I add the following dependencies to the `Aircraft Positions` application's *pom.xml*:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

NOTE

As previously mentioned, I include binders and drivers for both RabbitMQ and Kafka for planned future use, but only the RabbitMQ set—`spring-boot-starter-amqp` and `spring-cloud-stream-binder-rabbit`—are required for the current use case in order for Spring Cloud Stream (`spring-cloud-stream`) to use RabbitMQ.

I also add the two additional required entries to *pom.xml*. First, this goes into the `<properties></properties>` section, with the `java.version`:

```
<spring-cloud.version>2020.0.0-M5</spring-cloud.version>
```

Second is the Spring Cloud BOM information:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

A quick refresh of the project's dependencies and we're on to the next step.

Consuming aircraft positions

In order to retrieve and store messages listing current aircraft positions, only one small additional class is required:

```
@AllArgsConstructor
@Configuration
public class PositionRetriever {
    private final AircraftRepository repo;

    @Bean
    Consumer<List<Aircraft>> retrieveAircraftPositions() {
        return acList -> {
            repo.deleteAll();

            repo.saveAll(acList);

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

Like its `PositionReporter` counterpart in `PlaneFinder`, the `PositionRetriever` class is an `@Configuration` class in which I define a bean for use with Spring Cloud Stream: in this case, a `Consumer` of messages, each consisting of a `List` of one or more `Aircraft`. With each incoming message, the `Consumer` bean deletes all positions in the (in-memory) datastore, saves all incoming positions, and then prints all stored positions to the console for verification. Note that the last statement printing all positions to the console is optional; it's included only for confirmation as I develop the app.

Application properties

In order to provide the application the few remaining bits of information necessary to connect to the incoming stream of messages, I add the following entries to the *application.properties* file:

```
spring.cloud.stream.bindings.retrieveAircraftPositions-in-0.destination=
   aircraftpositions
spring.cloud.stream.bindings.retrieveAircraftPositions-in-0.group=
   aircraftpositions
spring.cloud.stream.bindings.retrieveAircraftPositions-in-0.binder=
   rabbit
```

As with `PlaneFinder`, the channel is defined by concatenating the following, separated by a hyphen (-):

- The bean name, in this case, a `Consumer<T>` bean
- `in`, since consumers only consume and thus have only input(s)
- A number between `0` and `7` inclusive, supporting up to eight inputs

The `destination` and `binder` properties match those of `PlaneFinder` because the `Aircraft Positions` application must point to the same destination as input that `PlaneFinder` used as output and because to do so, both must be using the same messaging platform—in this case, RabbitMQ. The `group` property is new, though.

For any kind of `Consumer` (including the receiving portion of a `Function<T, R>`), one can specify a `group`, but it isn't required; in fact, including or omitting `group` forms a starting point for a particular routing pattern.

If a message-consuming application doesn't specify a group, the RabbitMQ binder creates a randomized unique name and assigns it, and the consumer, to an auto-delete queue within the RabbitMQ instance or cluster. This results in each generated queue being serviced by one—and only one—consumer. Why is this important?

Whenever a message arrives at a RabbitMQ exchange, a copy is routed automatically to all queues assigned to that exchange by default. If an exchange has multiple queues, the same message is sent to every queue in what's referred to as a *fan-out pattern*, a useful capability when each message must be delivered to numerous destinations to satisfy various requirements.

If an application specifies a consumer group to which it belongs, that group name is used to name the underlying queue within RabbitMQ. When multiple applications specify the same `group` property and thus connect to the same queue, together those applications fulfill the competing consumer pattern in which each message arriving in the designated queue is processed by only one of the consumers. This allows the number of consumers to scale to accommodate varying volumes of messages.

NOTE

It is also possible to employ partitioning and routing keys for even finer-grained and flexible routing options, if needed.

Specifying the `group` property for this application enables scaling, should multiple instances be needed to keep pace with the flow of arriving messages.

Contacting the Controller

Since the `Consumer` bean automatically checks for and processes messages automatically, the `PositionController` class and its `getCurrentAircraftPositions()` method become dramatically leaner.

All references to `WebClient` can be removed, since getting a list of current positions is now only a matter of retrieving the current contents of the repository. The streamlined class now looks like this:

```
@RequiredArgsConstructor
@Controller
public class PositionController {
    @NonNull
    private final AircraftRepository repository;

    @GetMapping("/aircraft")
    public String getCurrentAircraftPositions(Model model) {
        model.addAttribute("currentPositions", repository.findAll());
        return "positions";
    }
}
```

With that, all changes to both message-producer (the `PlaneFinder` app) and message-consumer (the `Aircraft Positions` app) are now complete.

NOTE

In order to use any external messaging platform, said platform must be running and accessible to the applications. I run a local instance of RabbitMQ using Docker; scripts for quick creation and startup/shutdown are provided in this book's associated repositories.

The results

After verifying that RabbitMQ is accessible, it's time to start the applications and verify everything works as expected.

Although it isn't a requirement to do so, I prefer to start the message-consuming application first so it's ready and waiting for messages to arrive. In this case, that means executing `Aircraft Positions` from my IDE.

Next, I start up the new and improved `PlaneFinder` application. This initiates the flow of messages to the `Aircraft Positions` application, as shown in the `Aircraft Positions` app's console. That's gratifying, but we can follow this path of success all the way to the end user as well.

Returning to the browser and accessing *localhost:8080*, we're presented with the landing page once again, and opting to *Click here*, are taken to the Positions Report. As before, the Positions Report is refreshed automatically and displays current aircraft positions; now however, those positions are pushed independently from `PlaneFinder` behind the scenes to the `Aircraft Positions` application, without first receiving an HTTP request for them, which brings the architecture one step closer to a fully event-driven system.

Creating Conversations with WebSocket

In its first iteration, the distributed system we created to query and display current aircraft positions was entirely pull-based. A user requested (or re-requested with a refresh) the latest positions from the browser, which passed the request to the `Aircraft Positions` application, which in turn relayed the request to the `PlaneFinder` application. Responses then were returned from one to the next, to the next. The last chapter segment replaced the midsection of our distributed system with an event-driven architecture. Now whenever `PlaneFinder` retrieves positions from the upstream radio device, it pushes those positions to a streaming platform pipeline and the `Aircraft Positions` app consumes them. The last mile (or kilometer, if you prefer) is still pull-based, however; updates must be requested via browser refresh, either manually or automatically.

Standard request-response semantics work brilliantly for numerous use cases, but they largely lack the ability for the responding "server" side to, independent of any request, initiate a transmission to the requestor. There are various workarounds and clever ways to satisfy this use case—each of which has its own pros and cons, and some of the best of which I discuss in subsequent chapters—but one of the more versatile options is WebSocket.

What Is WebSocket?

In a nutshell, WebSocket is a full-duplex communications protocol that connects two systems over a single TCP connection. Once a WebSocket connection is established, either party can initiate a transmission to the other, and the designated server application can maintain numerous client connections, enabling low-overhead broadcast and chat types of systems. WebSocket connections are forged from standard HTTP connections using the HTTP upgrade header, and once the handshake is complete, the protocol used for the connection shifts from HTTP to WebSocket.

WebSocket was standardized by the IETF in 2011, and by now every major browser and programming language supports it. Compared to HTTP requests and responses, WebSocket is extremely low overhead; transmissions don't have to identify themselves and the terms of their communication with each transmission, thus reducing WebSocket framing to a few bytes. With its full-duplex capabilities, the ability of a server to handle a multiple of the number of open connections other options can support, and its low overhead, WebSocket is a useful tool for developers to have in their toolbox.

Refactoring the Aircraft Positions Application

Although I refer to the `Aircraft Positions` application as a single unit, the *aircraft-positions* project comprises both the backend Spring Boot+Java application and the frontend HTML+JavaScript functionality. During development, both portions execute in a single environment, usually the developer's machine. While they are built, tested, and deployed as a single unit to production settings as well, execution in production settings is divided as follows:

- Backend Spring+Java code is run in the cloud, including the template engine (if applicable) that generates final webpages to deliver to the end user.
- Frontend HTML+JavaScript—static and/or generated content—is displayed and run in the end user's browser, wherever that browser may be located.

In this section, I leave existing functionality intact and add the ability to the system to automatically display aircraft positions as they are reported via a live feed. With a WebSocket connection in place between frontend and backend applications, the backend app is free to push updates to the end user's browser and update the display automatically, with no need to trigger a page refresh.

Additional dependencies

To add WebSocket capabilities to the `Aircraft Positions` application, I need add only a single dependency to its *pom.xml*:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

A quick refresh of the project's dependencies and we're on to the next step.

Handling WebSocket connections and messages

Spring offers a couple of different approaches for configuring and using WebSocket, but I recommend following the clean lines of a direct implementation based on the `WebSocketHandler` interface. Owing to the frequency of requirements for exchanging text-based, i.e., nonbinary, information, there is even a `TextWebSocketHandler` class. I build on that here:

```
@RequiredArgsConstructor
@Component
public class WebSocketHandler extends TextWebSocketHandler {
    private final List<WebSocketSession> sessionList = new ArrayList<>();
    @NonNull
    private final AircraftRepository repository;

    public List<WebSocketSession> getSessionList() {
        return sessionList;
    }

    @Override
    public void afterConnectionEstablished(WebSocketSession session)
            throws Exception {
        sessionList.add(session);
        System.out.println("Connection established from " + session.toStrin
            " @ " + Instant.now().toString());
    }

    @Override
    protected void handleTextMessage(WebSocketSession session,
            TextMessage message) throws Exception {
        try {
            System.out.println("Message received: '" +
                message + "', from " + session.toString());

            for (WebSocketSession sessionInList : sessionList) {
                if (sessionInList != session) {
                    sessionInList.sendMessage(message);
                    System.out.println("--> Sending message '"
                        + message + "' to " + sessionInList.toString());
                }
            }
        } catch (Exception e) {
                System.out.println("Exception handling message: " +
            e.getLocalizedMessage());
        }
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session,
            CloseStatus status) throws Exception {
        sessionList.remove(session);
        System.out.println("Connection closed by " + session.toString() +
            " @ " + Instant.now().toString());
    }
}
```

The preceding code implements two of the `WebSocketHandler` interface's methods, `afterConnectionEstablished` and `afterConnectionClosed`, to maintain a `List` of active `WebSocketSession` and `log connections` and `disconnections`. I also implement `handleTextMessage`, broadcasting any incoming message to all other active sessions. This single class provides the WebSocket capability for the backend, ready to be activated when aircraft positions are received from `PlaneFinder` via RabbitMQ.

Broadcasting aircraft positions to WebSocket connections

In its previous iteration, the `PositionRetriever` class consumed aircraft position lists received via RabbitMQ messages and stored them in the in-memory H2 database. I build on that now by replacing the logging confirmation `System.out::println` call with a call to a new `sendPositions()` method, whose purpose is to use the newly added `@Autowired` `WebSocketHandler` bean to send the latest list of aircraft positions to all WebSocket-connected clients:

```
@AllArgsConstructor
@Configuration
public class PositionRetriever {
    private final AircraftRepository repository;
    private final WebSocketHandler handler;

    @Bean
    Consumer<List<Aircraft>> retrieveAircraftPositions() {
        return acList -> {
            repository.deleteAll();

            repository.saveAll(acList);

            sendPositions();
        };
    }

    private void sendPositions() {
        if (repository.count() > 0) {
            for (WebSocketSession sessionInList : handler.getSessionList()) {
                try {
                    sessionInList.sendMessage(
                        new TextMessage(repository.findAll().toString())
                    );
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Now that we have WebSocket configured properly and have a way for the backend to broadcast aircraft positions to connected WebSocket clients as soon as a new position list is received, the next step is to provide a way for the backend application to listen for and accept connection requests. This is accomplished by registering the `WebSocketHandler` created earlier via the `WebSocketConfigurer` interface and annotating the new `@Configuration` class with `@EnableWebSocket` to direct the application to process WebSocket requests:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {
    private final WebSocketHandler handler;

    WebSocketConfig(WebSocketHandler handler) {
        this.handler = handler;
    }

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry
        registry.addHandler(handler, "/ws");
    }
}
```

In the `registerWebSocketHandlers(WebSocketHandlerRegistry registry)` method, I tie the `WebSocketHandler` bean created earlier to the endpoint *ws://<hostname:hostport>/ws*. The application will listen on this endpoint for HTTP requests with WebSocket upgrade headers and act accordingly when one is received.

NOTE

If HTTPS is enabled for your application, *wss://* (WebSocket Secure) would be used in place of *ws://*.

WebSocket in back, WebSocket in front

With the backend work done, it's time to collect the payoff in the frontend functionality.

To create a simple example of how WebSocket enables the backend app to push updates unprompted by the user and their browser, I create the following file with a single HTML division and label and a few lines of JavaScript and place it in the project's *src/main/resources/static* directory along with the existing *index.html*:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <title>Aircraft Position Report (Live Updates)</title>
    <script>
        var socket = new WebSocket('ws://' + window.location.host + '/ws');

        socket.onopen = function () {
            console.log(
              'WebSocket connection is open for business, bienvenidos!');
        };

        socket.onmessage = function (message) {
            var text = "";
            var arrAC = message.data.split("Aircraft");
            var ac = "";

            for (i = 1; i < arrAC.length; i++) {
                ac = (arrAC[i].endsWith(", "))
                    ? arrAC[i].substring(0, arrAC[i].length - 2)
                    : arrAC[i]

                text += "Aircraft" + ac + "\n\n";
            }

            document.getElementById("positions").innerText = text;
        };

        socket.onclose = function () {
            console.log('WebSocket connection closed, hasta la próxima!');
        };
    </script>
</head>
<body>
<h1>Current Aircraft Positions</h1>
<div style="border-style: solid; border-width: 2px; margin-top: 15px;
        margin-bottom: 15px; margin-left: 15px; margin-right: 15px;">
    <label id="positions"></label>
</div>
</body>
</html>
```

As short as this page is, it could be shorter. The `socket.onopen` and `socket.onclose` function definitions are logging functions that could be omitted, and `socket.onmessage` could almost certainly be refactored by someone with actual JavaScript chops and the desire to do so. These are the key bits:

- The defined division and label in the HTML at bottom
- The `socket` variable that establishes and references a WebSocket connection

- The `socket.onmessage` function that parses the aircraft position list and assigns the reformatted output to the HTML “positions” label’s `innerText`

Once we rebuild and execute the project, it is of course possible to simply access the *wspositions.html* page directly from the browser. This is a poor way to create an application for actual users, though—providing no way to access a page and its functionality unless they know its location and enter it manually into the address bar—and it does nothing to set the table for upcoming chapters’ expansions to this example.

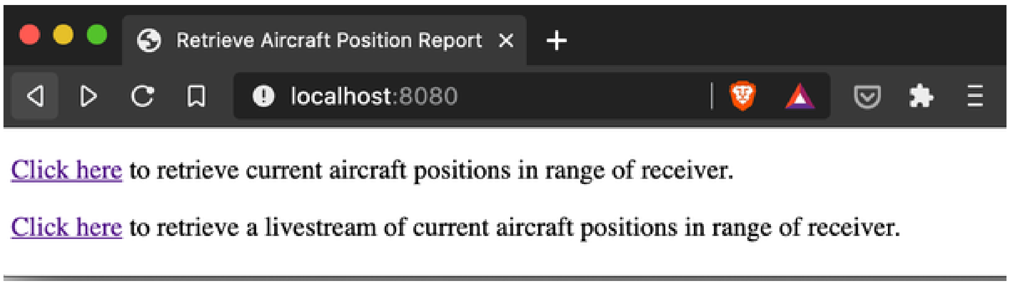
Keeping it simple for the time being, I add another line to the existing *index.html* to allow the user to navigate to the *wspositions.html* WebSocket-driven page in addition to the existing one:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Retrieve Aircraft Position Report</title>
</head>
<body>
    <p><a href="/aircraft">Click here</a> to retrieve current aircraft posi
        in range of receiver.</p>
    <p><a href="/wspositions.html">Click here</a> to retrieve a livestream
        current aircraft positions in range of receiver.</p>
</body>
</html>
```

With frontend work now complete, it’s time to test the WebSocket waters.

The results

From the IDE, I launch the `Aircraft Positions` application and `PlaneFinder`. Opening a browser window, I access the frontend application at *localhost:8080*, as shown in Figure 7-3.

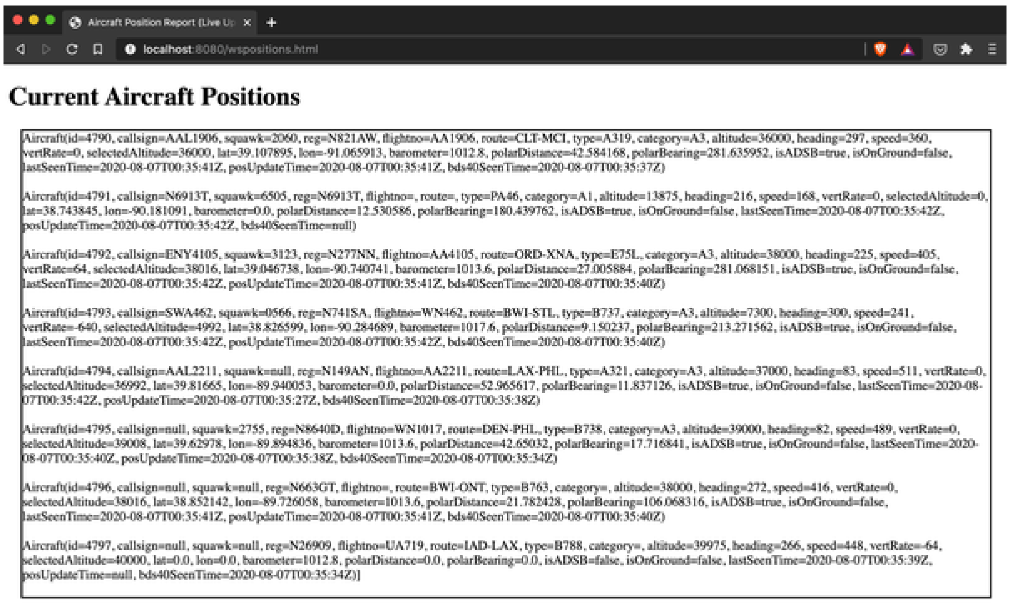


Click here to retrieve current aircraft positions in range of receiver.

Click here to retrieve a livestream of current aircraft positions in range of receiver.

Figure 7-3. Aircraft Positions landing page, now with two options

From the still rather rudimentary landing page, choosing the second option—*Click here* to retrieve a livestream of current aircraft positions in range of receiver—produces the *wspositions.html* page and results similar to those shown in Figure 7-4.


Aircraft Position Report (Live Up
localhost:8080/wspositions.html

Current Aircraft Positions

```
Aircraft(id=4790, callsign=AAL1906, squawk=2060, reg=N821AW, flightno=AA1906, route=CLT-MCI, type=A319, category=A3, altitude=36000, heading=297, speed=360,
vertRate=0, selectedAltitude=36000, lat=39.107895, lon=-91.065913, barometer=1012.8, polarDistance=42.584168, polarBearing=281.635952, isADSB=true, isOnGround=false,
lastSeenTime=2020-08-07T00:35:41Z, posUpdateTime=2020-08-07T00:35:41Z, bds40SeenTime=2020-08-07T00:35:37Z)

Aircraft(id=4791, callsign=N6913T, squawk=6505, reg=N6913T, flightno=, route=, type=PA46, category=A1, altitude=13875, heading=216, speed=168, vertRate=0, selectedAltitude=0,
lat=38.743845, lon=-90.181091, barometer=0.0, polarDistance=12.530586, polarBearing=180.439762, isADSB=true, isOnGround=false, lastSeenTime=2020-08-07T00:35:42Z,
posUpdateTime=2020-08-07T00:35:42Z, bds40SeenTime=null)

Aircraft(id=4792, callsign=ENY4105, squawk=3123, reg=N277NN, flightno=AA4105, route=ORD-XNA, type=E75L, category=A3, altitude=38000, heading=225, speed=405,
vertRate=64, selectedAltitude=38016, lat=39.046738, lon=-90.740741, barometer=1013.6, polarDistance=27.005884, polarBearing=281.068151, isADSB=true, isOnGround=false,
lastSeenTime=2020-08-07T00:35:42Z, posUpdateTime=2020-08-07T00:35:41Z, bds40SeenTime=2020-08-07T00:35:40Z)

Aircraft(id=4793, callsign=SWA462, squawk=0566, reg=N741SA, flightno=WN462, route=BWI-STL, type=B737, category=A3, altitude=7300, heading=300, speed=241,
vertRate=-640, selectedAltitude=4992, lat=38.826599, lon=-90.284689, barometer=1017.6, polarDistance=9.150237, polarBearing=213.271562, isADSB=true, isOnGround=false,
lastSeenTime=2020-08-07T00:35:42Z, posUpdateTime=2020-08-07T00:35:42Z, bds40SeenTime=2020-08-07T00:35:40Z)

Aircraft(id=4794, callsign=AAL2211, squawk=null, reg=N149AN, flightno=AA2211, route=LAX-PHL, type=A321, category=A3, altitude=37000, heading=83, speed=511, vertRate=0,
selectedAltitude=36992, lat=39.81665, lon=-89.940053, barometer=0.0, polarDistance=52.965617, polarBearing=11.837126, isADSB=true, isOnGround=false, lastSeenTime=2020-08-
07T00:35:42Z, posUpdateTime=2020-08-07T00:35:27Z, bds40SeenTime=2020-08-07T00:35:38Z)

Aircraft(id=4795, callsign=null, squawk=2755, reg=N8640D, flightno=WN1017, route=DEN-PHL, type=B738, category=A3, altitude=39000, heading=82, speed=489, vertRate=0,
selectedAltitude=39008, lat=39.62978, lon=-89.894836, barometer=1013.6, polarDistance=42.65032, polarBearing=17.716841, isADSB=true, isOnGround=false, lastSeenTime=2020-
08-07T00:35:40Z, posUpdateTime=2020-08-07T00:35:38Z, bds40SeenTime=2020-08-07T00:35:34Z)

Aircraft(id=4796, callsign=null, squawk=null, reg=N663GT, flightno=, route=BWI-ONT, type=B763, category=, altitude=38000, heading=272, speed=416, vertRate=0,
selectedAltitude=38016, lat=38.852142, lon=-89.726058, barometer=1013.6, polarDistance=21.782428, polarBearing=106.068316, isADSB=true, isOnGround=false,
lastSeenTime=2020-08-07T00:35:41Z, posUpdateTime=2020-08-07T00:35:41Z, bds40SeenTime=2020-08-07T00:35:40Z)

Aircraft(id=4797, callsign=null, squawk=null, reg=N26909, flightno=UA719, route=IAD-LAX, type=B788, category=, altitude=39975, heading=266, speed=448, vertRate=-64,
selectedAltitude=40000, lat=0.0, lon=0.0, barometer=1012.8, polarDistance=0.0, polarBearing=0.0, isADSB=false, isOnGround=false, lastSeenTime=2020-08-07T00:35:39Z,
posUpdateTime=null, bds40SeenTime=2020-08-07T00:35:34Z)]
```

Figure 7-4. Aircraft Position report with live updates via WebSocket

It's a trivial exercise to convert the database record format shown with JSON, and just a bit more involved to dynamically populate a table with results received live from the backend application via WebSocket. Please refer to this book's code repositories for examples.

TIP

It's perfectly fine to build and run both the `PlaneFinder` and `Aircraft Positions` applications from the command line; while I do so on occasion, for most build/run cycles, I find it much faster to run (and debug) directly from within the IDE.

Summary

Nearly every application must interact with end users or other applications in some manner to provide real utility, and that requires useful and efficient means of interaction.

This chapter introduced view technologies—template languages/tags like Thymeleaf and engines that process them—and how Spring Boot uses them to create and deliver functionality to an end user's browser. Also introduced was how Spring Boot handles static content like standard HTML along with JavaScript that can be delivered directly without processing by template engines. The chapter's first project iteration showed examples of both with a Thymeleaf-driven application that retrieved and displayed aircraft positions within range at the time of the request, an entirely pull-based model.

The chapter next showed how to harness the power of messaging platforms from Spring Boot using Spring Cloud Stream and RabbitMQ. The `PlaneFinder` application was refactored to push a list of current aircraft positions, each time retrieved from the upstream device, and the `Aircraft Positions` app was modified to accept new aircraft position listings as they arrived via the RabbitMQ pipeline. This replaced the pull-based model between the two applications with a push-based one, making the backend functionality of the `Aircraft Positions` app event-driven. The front end functionality still required a refresh (either manual or hard-coded) to update results shown to the user.

Finally, implementing a WebSocket connection and handler code within backend and frontend components of the `Aircraft Positions` application enabled the Spring+Java backend app to push aircraft position updates *as they are received* via a RabbitMQ pipeline from `PlaneFinder`. Position updates are shown live in a simple HTML+JavaScript page and require no update requests be issued by the end user or their browser, showcasing WebSocket's bidirectional nature, lack of required request-response pattern (or workaround), and low communication overhead.

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter7end* from the code repository.

The next chapter introduces reactive programming and describes how Spring is leading the development and advancement of numerous tools and technologies that make it one of the best possible solutions for numerous use cases. More specifically, I'll demonstrate how to use Spring Boot and Project Reactor to drive database access, integrate reactive types with view technologies like Thymeleaf, and take interprocess communication to unexpected new levels.