brief contents

1  ■  Introducing Spring: the dependency injection container 1
2  ■  Data persistence, ORM, and transactions 33
3  ■  Building web applications with Spring Web MVC 65
4  ■  Basic web forms 105
5  ■  Enhancing Spring MVC applications with Web Flow 134
6  ■  Authenticating users 173
7  ■  Authorizing user requests 209
8  ■  Communicating with users and customers 244
9  ■  Creating a rich-text comment engine 277
10 ■  Integration testing 306
11 ■  Building a configuration management database 338
12 ■  Building an article-delivery engine 392
13 ■  Enterprise integration 422
14 ■  Creating a Spring-based “site-up” framework 467
This chapter covers

- Shared database integration using Spring Data JPA
- Integrating web services with Spring Data REST
- Messaging via Spring Integration with RabbitMQ and JavaMail

An enterprise of any size might have hundreds of different software systems, such as monitoring tools, ticketing systems, collaboration platforms, and so forth. And if it appears that there isn’t much rhyme or reason to the specific mix of tools, that’s often because there isn’t. In a perfect world, there might be a fixed set of business processes, and the systems chosen to support those processes would play nicely together. But this isn’t a perfect world.

There are many reasons why the systems in an environment might be a jumbled mess. Among them:

- Business needs change. Those changes drive tool changes.
- Different teams in an organization have different tool preferences.
- A single vendor may offer a highly integrated tool suite, but the IT organization may prefer a best-of-breed tool strategy, or may prefer to keep multiple vendors in the game just to keep any one vendor from gaining too much leverage.
A whirlwind tour of Spring Integration

Throw in some vendor-sponsored lunches, events, and outings, and suddenly the systems start to look as if they were deliberately chosen not to work together!

Those are some of the realities of enterprise IT. And despite the difficulty, the need for systems integration is very much alive and well.

In this chapter you’ll imagine that you’re building a Spring-based help desk system to support an external-facing, internally developed customer portal. On the inbound side, you have the following requirements:

- Customers must be able to create tickets in a self-service fashion.
- The help desk system must have an internal-facing UI that lets support reps create tickets on behalf of customers who call the help desk on the phone.
- You need to support self-service ticket creation through a legacy email address that some customers still use even though it is no longer published.

On the outbound side, you’ll suppose that the help desk must send the customer a confirmation email regardless of whether the customer or a support rep creates the ticket. Figure 13.1 shows a conceptual overview of your eventual goal in this chapter.

A real help desk system has lots of other functionality, such as workflows, reporting, and so on. But we’re more concerned with showing how to perform integrations than with creating an actual ticketing system, so we’re sticking to a basic structure.

Although you’ll use a variety of integration technologies—including Spring Data REST, Spring HATEOAS, Spring Integration, Spring AMQP, Spring Rabbit, and RabbitMQ—you’re going to use Spring Integration (SI) particularly heavily. That being the case, let’s pause for a quick tour. Although it’s outside the scope of this chapter to present SI comprehensively,1 a better picture will emerge as you work through the recipes.

A whirlwind tour of Spring Integration

SI is an implementation of the patterns described in Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf (Addison-Wesley, 2003). EIP is a comprehensive catalog of patterns helpful in getting applications that weren’t initially designed to work...

---

1 Please see Spring Integration in Action by Mark Fisher, et al., (Manning, 2012) for a more detailed treatment of Spring Integration.
together to do so. Note that we assume some familiarity with these patterns and the integration domain in general; see www.eaipatterns.com/ for more background if you require it.

Something interesting about SI is the way it uses dependency injection. The typical Spring application uses DI in what might be called a vertical fashion: the application is organized as a set of layers, and you inject beans at layer $n$ into beans at layer $n+1$. For example, you inject DAOs into service beans and service beans into web controllers.

SI applies dependency injection primarily in a horizontal way. The idea is to build an integration layer just above an application’s services and implement the integration layer in the pipes and filters architectural style. In SI, the pipes are called *channels* and the filters are called *endpoints*, and the overall pipeline is a messaging system supporting the integration of connected services. Endpoints perform message routing and processing, whereas channels convey messages from endpoint to endpoint. You can build an entire messaging system out of channels and endpoints by injecting channels into endpoints. Figure 13.2 shows how this works.

This horizontal use of DI is entirely compatible with the vertical use. Especially when you’re integrating internally developed apps, it’s sometimes helpful to have *service activators* in the integration layer invoke your Java service beans directly. You link service activators to service beans through DI.

But in general, SI uses dependency injection as a way to build a horizontal layer of integration flows. Most of the communication with underlying services happens through the application’s coarse-grained external interfaces (for example, web services and messaging endpoints) rather than through the fairly fine-grained mechanism of Java DI.

Note that although SI provides ready-made implementations of the various EIP patterns, there are architecturally different approaches to applying those patterns, and SI isn’t prescriptive about it. You can use SI to implement a simple, in-process messaging

---

**Integration and services: an architectural perspective**

One of the things I (Willie) struggled with when I was first learning about integration was understanding just what it is that’s being integrated. It can get confusing because there are multiple ways of talking about services, and they often come up in a single conversation.

---

2 For more information on the pipes-and-filters architectural style, see Pipes and Filters, Enterprise Integration Patterns, www.eaipatterns.com/PipesAndFilters.html.
Integrating applications via a shared database

I won’t attempt a definition here, but a loose characterization will help. Services in an integration context refers to the coarse-grained interfaces that different apps/systems happen to expose to the world and that provide integration hooks. This could be SOAP/REST web services, messaging endpoints, HTML forms, file-based, email-based, and so forth. Obviously this is a permissive conception, and not one you would use to prescribe for a green field, service-oriented architecture. But for integration it’s appropriate because it’s sometimes the case that the only way to talk to an app is to post data into an HTML form that it provides.

Services in this sense refers to something larger than the service beans we often find in Spring-based applications, because (for example) RESTful endpoints live above the service beans. There are certainly cases where the integration layer has direct access to an application’s Java code, and so the integration layer might call that code directly, but this is a special instance. The more general case is an integration layer interacting with apps/systems through whatever interfaces they expose.

You’ll see in this chapter, for instance, how creating a ticket in the portal app causes the creation of a ticket in the help desk app, which in turn generates a confirmation email. These apps and systems were independently conceived, but you use integration to make them work together.

infrastructure that a given app uses to communicate with external resources. Alternatively, you can use it to communicate with an external message broker (ActiveMQ, RabbitMQ, SonicMQ, and so on), which usually provides better decoupling, flexibility, and resiliency. You’ll see examples of both of those in the recipes ahead.

The first recipe starts you off by illustrating a common form of application integration: a shared database.

13.1 Integrating applications via a shared database

PREREQUISITES
Recipe 2.6 Spring Data JPA overview

KEY TECHNOLOGIES
Database

Background
In many cases, multiple apps need to work with the same data. One app might capture leads, another might qualify the leads, and a third might present the leads to salespeople trying to close. Especially when the apps in question are developed internally, it’s often easiest and most appropriate to have the apps all work with a shared database.

Don’t laugh. There are major, revenue-critical systems that work exactly this way.

There are other possibilities as well. For instance, you could stand up a dedicated (that is, not directly attached to any existing app) integration bus based on Spring Integration. We won’t explore that here, but by the end of this chapter it should be obvious how to do it.
Problem
Integrate applications by having them work against a shared database.

Solution
For your current purposes, assume that your help desk and portal applications need to work with common customer and ticket data. Customers need to be able to create tickets using the portal, and those tickets need to show up in the help desk app for processing by support representatives.

The approach here, as noted, will be to use a shared database. This approach generally requires that you control the apps in question because you have to be able to make them agree on the database schema. But where that’s true, a shared database is an option to consider. At small scale, the coordination between apps on schema-related issues is generally manageable, and this allows you to avoid having to build out separate abstraction layers such as web services. See figure 13.3.

There isn’t anything too special about having two different apps use the same database. You have to pay attention to transactions, but that’s usually true for single apps. Point your apps at the same database, and you’re in business.

Your real goal in this recipe is to establish a code baseline for subsequent recipes. To that end, you use Spring Data JPA as described in recipe 2.6, rather than working directly with the Hibernate API. (You still use Hibernate as a JPA provider.) The reason is that you want to set the stage for using Spring Data REST in recipe 13.2, and this depends on Spring Data JPA. Feel free to review recipe 2.6 or look at the sample code.

One special case comes up in the context of shared database integrations, and it’s worth investigating.

GETTING A SINGLE APP TO WORK WITH MULTIPLE DATABASES
Even though this is a recipe about having multiple apps work with a shared database, sometimes that plays out via an app with a dedicated database, and the app needs data from another app’s database as well. Now you have an app that has to work with two databases.

In this case, the portal app owns the customer database, and the help desk app owns the ticket database. But the help desk app needs customer data so it can resolve customer usernames in the tickets to the customer’s full information (name, contact information,
Integrating applications via a shared database

and so on). So the help desk app will use the customer database in addition to its own ticket database. See figure 13.4.

Let’s discuss the help desk for a moment. One possibility for working with multiple databases is to use distributed transactions. Even though you’re only reading customer data, in theory transactions could be useful for proper isolation, setting lock modes, and setting timeouts. You might use them, for instance, to handle the case where the portal changes a customer username or deletes a customer entirely, either of which would break the soft association (based on usernames) between tickets and their customers.

But in fact there’s no worry here, because the portal doesn’t allow users to change their usernames, and it also doesn’t delete customers. (With customer data, you’d probably want soft deletes, where you keep the customer record but use a flag to indicate whether the customer has been deleted.) And even if you were for whatever reason to allow these things, they would presumably be insufficiently common to warrant the performance overhead of distributed transactions.

Instead, your help desk will use two different transaction managers: one for working with tickets and the other for working with customers. You haven’t seen that yet, so let’s look at how the help desk app does it. First, the following listing shows the help desk’s configuration for the ticket database.

Listing 13.1 beans-repo.xml: configuration for the ticket database

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:jee="http://www.springframework.org/schema/jee"
xmlns:jpa="http://www.springframework.org/schema/data/jpa"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/data/jpa
http://www.springframework.org/schema/data/jpa/spring-jpa-1.2.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.1.xsd">

  <jee:jndi-lookup id="helpDeskDS"
    jndi-name="jdbc/Sip13HelpDeskDS" resource-ref="true" />

  <bean id="entityManagerFactory"

Using JPA
class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean"
p:dataSource-ref="helpDeskDS">
<property name="packagesToScan">
 <list>
 <value>com.springinpractice.ch13.helpdesk.model</value>
 <value>com.springinpractice.ch13.sitemap.model</value>
 </list>
 </property>
<property name="persistenceProvider">
 <bean class="org.hibernate.ejb.HibernatePersistence" />
 </property>
<property name="jpaProperties">
 <props>
 <prop key="hibernate.dialect">
 org.hibernate.dialect.MySQL5Dialect
 </prop>
 <prop key="hibernate.show_sql">false</prop>
 </props>
 </property>
</bean>

<bean id="transactionManager"
 class="org.springframework.orm.jpa.JpaTransactionManager"
p:entityManagerFactory-ref="entityManagerFactory" />
<tx:annotation-driven transaction-manager="transactionManager" />

<jpa:repositories
 base-package="com.springinpractice.ch13.helpdesk.repo"
 entity-manager-factory-ref="entityManagerFactory"
 transaction-manager-ref="transactionManager" />

<jpa:repositories
 base-package="com.springinpractice.ch13.sitemap.repo"
 entity-manager-factory-ref="entityManagerFactory"
 transaction-manager-ref="transactionManager" />

</beans>

You’re using Spring Data JPA, so you set up a factory for JPA entity managers at 1. At 2 you’re explicit about the entity manager factory (EMF) because you want to avoid ambiguities with the EMF you’re about to create in the next listing. Similarly for 3. You’re also explicit at 4 and 5 where you define the Spring Data JPA repositories. Spring Data JPA will apply the specified EMF and transaction manager to the repositories it generates. Your single transaction manager can handle multiple repositories just fine, because repositories correspond to tables, not to databases.5

The help desk app has a repository configuration file for the customer database as well.

5 The JtaTransactionManager can handle multiple databases in support of distributed transactions, but you aren’t using that here.
Integrating applications via a shared database

Listing 13.2 beans-repo-portal.xml: configuration for the customer database

```xml
<?xml version="1.0" encoding="UTF-8"?
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:jee="http://www.springframework.org/schema/jee"
 xmlns:jpa="http://www.springframework.org/schema/data/jpa"
 xmlns:p="http://www.springframework.org/schema/p"
 xmlns:tx="http://www.springframework.org/schema/tx"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=" http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
 http://www.springframework.org/schema/data/jpa
 http://www.springframework.org/schema/data/jpa/spring-jpa-1.2.xsd
 http://www.springframework.org/schema/jee
 http://www.springframework.org/schema/jee/spring-jee-3.1.xsd
 http://www.springframework.org/schema/tx
 http://www.springframework.org/schema/tx/spring-tx-3.1.xsd">
<jee:jndi-lookup id="portalDS"
 jndi-name="jdbc/Sip13PortalDS" resource-ref="true" />
<bean id="portalEMF"
 class="org.springframework.orm.jpa.
➥ LocalContainerEntityManagerFactoryBean"
 p:dataSource-ref="portalDS"
 p:packagesToScan="com.springinpractice.ch13.helpdesk.portal.model">
 <property name="persistenceProvider">
  <bean class="org.hibernate.ejb.HibernatePersistence" />
 </property>
 <property name="hibernate.show_sql">false</property>
 </bean>
<bean id="portalTxManager"
 class="org.springframework.orm.jpa.JpaTransactionManager"
 p:entityManagerFactory-ref="portalEMF" />
<tx:annotation-driven transaction-manager="portalTxManager" />
<jpa:repositories
  base-package="com.springinpractice.ch13.helpdesk.portal.repo"
  entity-manager-factory-ref="portalEMF"
  transaction-manager-ref="portalTxManager" />
</beans>
```

In the previous listing you’re explicit about the EMF and transaction manager, just as you were in listing 13.1.

Sometimes there’s only one database, and all the apps use it, but the configurations show how to deal with multi-database scenarios where distributed transactions aren’t a concern.
Chapter 13  Enterprise integration

Configuring and Running the Applications

You’re dealing with two separate applications here, and each has its own configuration location and configuration files. The configuration approach is essentially the same as the one the other chapters use (the one from appendix A), but instead of a single sip13 folder, you’ll need a sip13/helpdesk folder for the help desk configuration files and a sip13/folder for the portal configuration files.

Because you’re dealing with two applications here, the URLs are different than other URLs in the book:

- Portal application—http://localhost:8180/portal/

Run the apps, and try things out. You should see that they’re both sharing the portal’s customer data.

Discussion

In this recipe we looked at the common approach of using a shared database to integrate multiple applications. This didn’t require any special technology beyond what the database natively provides with respect to transaction management.

Despite its simplicity, it’s important to consider reasons why you might choose not to adopt this approach, even initially:

- A shared database works well enough at small scale, but as you add applications it becomes increasingly difficult to coordinate changes. One app may need a schema change that would break other apps, and so the change can’t occur until all apps are ready for it.
- Depending on the database technology, it may be difficult to scale out as you add apps.
- The shared database quickly becomes a single point of failure.
- By the time you decide to pursue a different integration approach, the apps are generally tightly coupled to the database.
- If you don’t control the apps in question, then a shared database probably isn’t an option for you in the first place.
- Even if you do control the apps, there’s a good chance that you have third-party apps that also need to work with the same data. So you may need to come up with an integration solution that doesn’t require a shared database anyway.

Web services are a popular and effective solution for some of the challenges described. They make it possible to decouple the client view of the data from the details of the actual database implementation, which provides needed flexibility. The next recipe shows how to integrate applications using RESTful web services.

13.2 Decoupling applications with RESTful web services

Prerequisites

Integrating applications via a shared database
KEY TECHNOLOGIES
Spring Data JPA, Spring Data REST, Spring HATEOAS

Background
In recipe 13.1 you saw that although shared database integrations may be simple at small scale, the approach can quickly run into both development and operational issues with increasing scale. Web services are a battle-tested technique for decoupling clients from the data they use. Originally SOAP-based web services were the “official” approach, but over time the growing consensus was that SOAP was too heavyweight for many purposes, and so the REST-based approach to web services took the lead. But SOAP or REST, the idea is to create an abstraction layer in front of core capabilities and data so the owners of those capabilities and data can make back-end changes without breaking their clients and without being held hostage by those clients (that is, being prevented from making changes).

With the REST approach in particular, service designers try to minimize the amount of knowledge that clients must have to work with the service. For example, REST’s Hypermedia as the Engine of Application State (HATEOAS) principle says that clients shouldn’t require anything beyond a general knowledge of working with hypermedia-based systems. This constraint, properly observed, further decouples clients from the services they use.

Problem
Integrate applications without the tight coupling entailed by shared database integrations.

Solution
This recipe takes a first step in the direction of decoupling your apps by eliminating their common dependency on the details of the shared customer database schema. You’ll learn how to do the following:

- Use Spring Data REST to implement (as part of the portal app) a RESTful web service API in front of your customer data
- Use Spring HATEOAS to implement data transfer objects
- Use RestTemplate to implement (as part of the help desk app) a client for your web service

In the sample code we introduce a dependency in the other direction as well: the portal needs to get data (ticket statuses, ticket categories) from the help desk’s ticket database to allow customers to create self-service tickets through the portal UI. See figure 13.5 for the updated, point-to-point integration architecture.

Because both directions are entirely symmetrical, we’ll cover only one direction here: the one where the help desk calls the portal web service API to get customer data. Refer to the sample code if you want to see the other direction as well.

---

CHAPTER 13 Enterprise integration

Figure 13.5 The point-to-point integration architecture for this recipe. The apps hide their databases from one another but expose RESTful web service APIs for data access.

You’ll begin by implementing the web service on the portal app.

IMPLEMENTING A RESTFUL WEB SERVICE USING SPRING DATA REST

Spring Data REST (SDR) is a relatively recent addition to the Spring family. It builds on Spring Data JPA (SDJ) to expose SDJ repositories through a RESTful, JSON-based web service API. SDR does this using an exporter servlet that knows how to interpret special REST annotations declared on the SDJ repository interfaces.

Let’s look first at the portal app’s single repository interface, UserRepository. This is the repo for customer data.

Listing 13.3 UserRepository: SDJ repository annotated for exporting by SDR

```java
package com.springinpractice.ch13.portal.repo;
import java.util.Collection;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.repository.annotation.RestResource;
import com.springinpractice.ch13.portal.model.User;

@RestResource(path = "users")
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(@Param("username") String username);
    List<User> findByUsernameIn(@Param("username") Collection<String> usernames);
}
```

At 1 you use the SDR @RestResource annotation to specify a path to the resource, relative to the servlet path. You’re specifying "users" because otherwise SDR defaults to "user" (based on UserRepository), which you don’t want. At 2 you attach an SDR path to your custom SDJ finder query. You specify its HTTP parameter at 3. You do the same thing at 4 and 5 for a collection-driven custom finder query.

---

7 At the time of this writing, it’s still a release candidate, so it may change a bit by the time you read this.
You’re almost done. All that remains is to define the exporter servlet in web.xml.

**Listing 13.4 Defining RepositoryRestExporterServlet in web.xml**

```xml
<web-app ...>
...
<servlet>
   <servlet-name>api</servlet-name>
   <servlet-class>org.springframework.data.rest.webmvc.
   RepositoryRestExporterServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>api</servlet-name>
   <url-pattern>/api/*</url-pattern>
</servlet-mapping>
</web-app>

That’s all there is to it. Let’s look at what it does. First, start up the portal app. Now go to http://localhost:8180/portal/api/users:

```json
{
   "links" : [ {
      "rel" : "users.search",
      "href" : "http://localhost:8180/portal/api/users/search"
   } ],
   "content" : [ {
      "links" : [ {
         "rel" : "self",
         "href" : "http://localhost:8180/portal/api/users/1"
      } ],
      "lastName" : "Jenson",
      "username" : "paul",
      "email" : "paul@example.com",
      "firstName" : "Paul"
   }, {
      "links" : [ {
         "rel" : "self",
         "href" : "http://localhost:8180/portal/api/users/2"
      } ],
      "lastName" : "Henshaw",
      "username" : "aimee",
      "email" : "aimee@example.com",
      "firstName" : "Aimee"
   } ],
   "page" : {
      "size" : 20,
      "totalElements" : 2,
      "totalPages" : 1,
      "number" : 1
   }
}
```
True to HATEOAS form, the resources specify links that you can follow to get further results. Let's see what happens when you hit the search endpoint:

```
{
  "links" : [ {
    "rel" : "users.find-by-username-in",
  }, {
    "rel" : "users.find-by-username",
    "href" : "http://localhost:8180/portal/api/users/search/find-by-username"
  } ],
  "content" : [ ]
}
```

Let's try finding by a username in a collection. If you do `http://localhost:8180/portal/api/users/search/find-by-username-in?username=aimee` you get:

```
{
  "links" : [ ],
  "content" : [ {
    "links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8180/portal/api/users/2"
    } ],
    "lastName" : "Henshaw",
    "username" : "aimee",
    "email" : "aimee@example.com",
    "firstName" : "Aimee"
  } ]
}
```

Try it with `http://localhost:8180/portal/api/users/search/find-by-username-in?username=aimee&username=paul`. You get the idea. For anybody who has ever implemented a RESTful web service, you can see that Spring Data REST is powerful, and that it saves a lot of effort.

To take advantage of the new web service, the help desk app will find it useful to have data transfer objects (DTOs) for data binding. You’ll use Spring HATEOAS to create those.

**IMPLEMENTING DATA TRANSFER OBJECTS USING SPRING HATEOAS**

SDR uses Spring HATEOAS behind the scenes to generate the JSON representation for requested resources. Besides the actual payload, Spring HATEOAS supports links, which you would expect because that’s a big part of the HATEOAS idea.

**NOTE** At the time of this writing, Spring HATEOAS has not been officially released, so there may be material changes by the time you read this.

On the client side, you want those links because SDR uses URIs for resource identification (again, no surprise) rather than the database IDs.
You can use Spring HATEOAS to implement the desired DTOs on the client side. Because both individual users and collections of users have associated link information, you need separate DTOs for each. The next listing shows the resource for an individual user, which the help desk refers to as a customer rather than a user because the help desk’s users are support representatives.

### Listing 13.5 CustomerResource.java: HATEOAS-oriented DTO

```
package com.springinpractice.ch13.helpdesk.integration.resource;
import org.springframework.hateoas.ResourceSupport;
public class CustomerResource extends ResourceSupport {
    public String username;
    public String firstName;
    public String lastName;
    public String email;

    public String getUsername() { return username; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String getEmail() { return email; }
    public String getFirstNameLastName() {
        return firstName + " " + lastName;
    }
}
```

You extend the Spring HATEOAS ResourceSupport class at 1, which does the heavy linking around generating links and such. The DTO’s fields need to be public in order for data binding to work, so that’s what you do at 2. JSPs don’t know how to deal with public fields, so you create getters at 3.

Here’s the same thing for a collection of customers.

### Listing 13.6 CustomerResources.java: another HATEOAS-oriented DTO

```
package com.springinpractice.ch13.helpdesk.integration.resource;
import org.springframework.hateoas.Resources;
public class CustomerResources extends Resources<CustomerResource> { }
```

Here all you do is extend the Resources class, specifying CustomerResource as the type argument. The last step is to implement a client for the help desk app.

**IMPLEMENTING A RESTFUL CLIENT USING RESTTEMPLATE**

In integration parlance, a gateway provides an application with an interface to the underlying messaging infrastructure without the application realizing it. Because you want the help desk to be able to get customer data from the portal without realizing that it’s making a web service call, you’ll create a PortalGateway interface for the help desk to use, along with an implementation that makes the web service call using RestTemplate and your resource DTOs. Here’s the interface.
Listing 13.7 PortalGateway: hides messaging details from the help desk app

```java
package com.springinpractice.ch13.helpdesk.integration.gateway;

import java.util.Collection;
    CustomerResource;

public interface PortalGateway {
    CustomerResource findCustomerByUsername(String username);
    Collection<CustomerResource> findCustomersByUsernameIn(
        Collection<String> usernames);
}
```

The following listing contains the implementation.

Listing 13.8 PortalGatewayImpl: uses RestTemplate to get customer data

```java
package com.springinpractice.ch13.helpdesk.integration.gateway.impl;

import java.util.Collection;
import org.springframework.web.client.RestTemplate;
    PortalGateway;
    CustomerResource;
    CustomerResources;

public class PortalGatewayImpl implements PortalGateway {
    private RestTemplate restTemplate;
    private String baseUrl;

    public PortalGatewayImpl(RestTemplate restTemplate, String baseUrl) {
        this.restTemplate = restTemplate;
        this.baseUrl...
    }

    @Override
    public CustomerResource findCustomerByUsername(String username) {
        String url = baseUrl +
            "/users/search/find-by-username?username={username};
        CustomerResources customers = restTemplate
            .getForObject(url, CustomerResources.class, username);
        CustomerResource customer =
            customers.getContent().iterator().next();
        return customer;
    }

    @Override
    public Collection<CustomerResource> findCustomersByUsernameIn(
        Collection<String> usernames) {
        StringBuilder builder = new StringBuilder(
            baseUrl + "/users/search/find-by-username-in?username={username}"
        );
        for (String username : usernames) {
            builder.append("username=");
        }
        
```

---

3. Gets customers

5. Builds URL

4. Gets single customer

2. API base URL

1. RestTemplate
Decoupling applications with RESTful web services

You use Spring’s RestTemplate 1 to invoke the portal’s web service API, located at the baseUrl 2. The gateway has two methods, corresponding to the two custom queries you implemented for the web service. The first one returns a single customer, which the help desk app uses to resolve a username to a customer record on a ticket details page. You use the CustomerResources 3 you wrote earlier to get the wrapper container, because Spring Data REST doesn’t know that customer usernames are unique. Then you get and return the desired CustomerResource 4.

The second method returns a collection of customers corresponding to a collection of usernames. You use this one for the ticket summary page, because there are a bunch of tickets, each with its own username. You build out the URL from scratch at 5 because there are arbitrarily many usernames (limited generally by paging the ticket summary). You drop the trailing ? or & at 6 and then get the CustomerResources 7. Finally you return the underlying collection of CustomerResource instances 8 to match the API signature.

That does it for the help desk app’s PortalGatewayImpl. Although we didn’t cover it here, the portal app has an analogous TicketGateway and TicketGatewayImpl to handle calls against the ticket service’s REST API. Take a look at the sample code for details.

One final note on gateways before we close the recipe. We mentioned earlier that gateways hide the details of the messaging approach from the app. Gateway implementations are part of the integration infrastructure, not part of the app. They allow you to change the integration approach with minimal disruption to the app. You’ll do exactly this in the next recipe.

RUNNING THE APPLICATIONS

Again, the URLs are

- **Help desk application**—http://localhost:8080/helpdesk/
- **Portal application**—http://localhost:8180/portal/

Note that you have to run both apps at the same time in order for either to work, because they make web service calls against one another. We’ll touch on this in the following discussion.

**Discussion**

The RESTful web service approach adopted in this recipe decouples the apps from the shared database from recipe 13.2. It also decouples the apps to some extent from one
another because now it’s possible for the apps to make back-end changes to their respective database schemas without impacting one another.

Unfortunately, the apps are still fairly tightly coupled to one another:

- From a development perspective, they need to coordinate with each other on changes to the web service API.
- From a configuration perspective, both apps have to know each other’s location.
- From an operational perspective, both apps have to be up for either app to work properly. So instead of a single point of failure, you have two.

In addition to the coupling that remains, there’s another problem. The point-to-point integration approach breaks down because it scales as $O(n^2)$ in the number of applications to be integrated. If you have more than a handful of applications to integrate, you’ll need to manage a lot of development, configuration, and operational linkages, as you can see in figure 13.6.

Recipe 13.3 shows how you can use a centralized messaging infrastructure to alleviate much of the pain.

### 13.3 Implementing a message bus using RabbitMQ and Spring Integration

**PREREQUISITES**

Recipe 13.2 Decoupling applications with RESTful web services

Familiarity with the integration domain and Enterprise Integration Patterns in particular; see www.eaipatterns.com/ for background.

**KEY TECHNOLOGIES**

SI, RabbitMQ, Spring Rabbit, Advanced Message Queuing Protocol (AMQP), Spring AMQP

**Background**

Recipe 13.2 explored the use of RESTful web service APIs as a way to decouple applications from one another. Web service APIs provide a layer of abstraction over underlying capabilities and data, which makes it easier to change the implementations without impacting clients. In addition, the RESTful approach supports decoupling by reducing the knowledge that clients must have of the services they consume.

We noted in the discussion that improvements are possible in two major areas. First, the apps still have to know quite a bit about each other from development, configuration, and operational perspectives. Second, the point-to-point integration strategy
scales poorly as you incorporate different apps. This recipe presents a broker-based approach that addresses both of these issues.

**Problem**

Further decouple your apps, and address scalability issues associated with the point-to-point integration strategy.

**Solution**

The solution is to use a centralized message broker to serve as the basis for your application integrations. Message brokers are specifically designed to address application integration, and they solve the previous issues as follows:

- Centralizing the integration infrastructure allows you to transform the $O(n^2)$ integration topology to an $O(n)$ topology. (Each app has a link to the central integration infrastructure.)
- The characteristics of message brokers promote decoupling through the use of asynchronous messaging with guaranteed delivery, well-known messaging endpoints, and so forth.
- Message brokers are generally reliable, runtime-configurable, and horizontally scalable, which helps with availability and performance. This largely mitigates the single-point-of-failure issues associated with being a central location in the architecture.

There are lots of options for message brokers, but you’ll use RabbitMQ, which implements the AMQP messaging protocol. The advantage over the Java Message Service (JMS) API is that using a protocol decouples messaging clients from the broker. With JMS, the clients are Java clients (although any given broker has APIs for other platforms as well). With AMQP, any platform with an AMQP client can communicate with the broker, in much the same way that any web browser can communicate with any web server, regardless of the client and server platforms. Because most platforms have AMQP clients,9 AMQP has outstanding interoperability.

We won’t go into the details of RabbitMQ; fortunately it’s fast and easy to set up a development instance.9 You can also consult *RabbitMQ in Action* by Alvaro Videla and Jason J.W. Williams (Manning, 2012) for further information.

Recall from the previous recipe that you used gateways to hide the messaging system from the help desk and portal apps. In this recipe, you’ll realize the advantage of that approach: you’ll throw away the point-to-point REST implementations entirely and replace them with SI–generated proxies that use AMQP to speak to RabbitMQ.10

---

8 Java, Ruby, Python, .NET, Perl, PHP, C/C++, Erlang, Lisp, Haskell . . .
9 See [www.rabbitmq.com/download.html](http://www.rabbitmq.com/download.html) for download and installation instructions.
10 Although it would be possible to use Spring Integration to bridge web services to RabbitMQ, here there’s no point. There are fewer moving parts if you remove the web services and connect the apps directly to the broker. In other situations it might be desirable to keep the web services around.
Figure 13.7 shows the goal for this recipe. Let’s get started by looking at message buses and canonical data models.

**MESSAGE BUSES AND CANONICAL DATA MODELS**

You’re going to use RabbitMQ to implement the *message bus* integration pattern. The idea behind this pattern is to provide a central medium through which applications can communicate with one another. Conceptually it’s based on the hardware bus concept: plug in, and you’re good to go. Hohpe and Woolf define a message bus as follows:

*A Message Bus is a combination of a Canonical Data Model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces.*

—*Enterprise Integration Patterns*, p. 139

You’ll use message queues as the shared set of interfaces. But what’s missing so far is the so-called canonical data model (CDM), which is the *lingua franca* that allows you to get away with \( O(n) \)—or even \( O(1) \)—message translations instead of \( O(n^2) \). In the previous recipe, the two apps had their own data representations. Now you’ll standardize those by creating a separate Maven module for the CDM.

In real life there are sometimes significant business, technical, and organizational challenges surrounding the creation of a CDM, but you can ignore those because you’re lucky enough to have a simple data model. You’ll use XML for your format because it’s widely supported, although JSON would be another plausible option. Ideally you’d create XML schemas for the model, but you won’t mess around with that here. Instead you’ll create new DTOs (the Spring HATEOAS DTOs are more oriented
Implementing a message bus using RabbitMQ and Spring Integration

around RESTful web services), define XML bindings, and treat the implied schema as constituting your CDM.

You have a handful of message types, but it will suffice to look at one. The following listing shows the DTO for tickets.

```java
package com.springinpractice.ch13.cdm;
import java.util.Date;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
public class Ticket {
    private TicketCategory category;
    private TicketStatus status;
    private String description;

    public TicketCategory getCategory() { return category; }
    public void setCategory(TicketCategory category) {
        this.category = category;
    }
    public TicketStatus getStatus() { return status; }
    public void setStatus(TicketStatus status) { this.status = status; }
    public String getDescription() { return description; }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getCreatedBy() { return createdBy; }
    public void setCreatedBy(String createdBy) {
        this.createdBy = createdBy;
    }
    public Date getDateCreated() { return dateCreated; }
    public void setDateCreated(Date dateCreated) {
        this.dateCreated = dateCreated;
    }
}
```

There isn’t much to say here. It’s a bare-bones DTO with some JAXB annotations to bind it to the CDM’s XML representation.

You have DTOs for other message types as well, such as ticket categories, ticket statuses, customers, and so forth. Consult the sample code for details.
Now that you have a CDM in place along with a central set of DTOs, you need to make an interesting design decision. One possibility is for the existing apps to continue using their existing data models, and perform translations as messages enter and exit the bus. The other is for apps to adopt the central DTOs as their own data model, at least in cases where you have control over that (for example, internally developed apps).

In this case the choice is fairly clear because the Spring HATEOAS DTOs are more oriented to support RESTful web services. The benefit is that you can avoid message translation between apps. You do need to modify the gateways to use the new DTOs, so let’s do that now.

**REVISITING THE GATEWAY INTERFACES**

In recipe 13.3 you worked on the help desk side with the PortalGateway. For variety, this time you’ll work on the portal side with the TicketGateway.

It happens that the gateway interfaces are slightly leaky: through the DTOs they use, they expose the fact that you’re designing for Spring Data REST-based implementations with URIs instead of database IDs.

Let’s replace the Spring HATEOAS DTOs with the ones you created for the CDM. The next listing presents the new TicketGateway.

```java
package com.springinpractice.ch13.portal.integration.gateway;
import com.springinpractice.ch13.cdm.Ticket;
import com.springinpractice.ch13.cdm.TicketCategory;
import com.springinpractice.ch13.cdm.TicketStatus;
public interface TicketGateway {
    void createTicket(Ticket ticket);
    TicketStatus findOpenTicketStatus();
    TicketCategoryList findTicketCategories();
    TicketCategory findTicketCategory(Long id);
}
```

See the sample code for a similar treatment of the PortalGateway. Let’s turn now to the gateway implementations.

**REIMPLEMENTING THE PORTAL’S TICKETGATEWAY USING SPRING INTEGRATION**

SI allows you to implement gateways dynamically. SI allows you to build integration logic that allows the portal app to send requests to other systems, and also to respond to requests from other systems. You can of course do the same thing for the help desk

---

11 HATEOAS is a general architectural principle and might make sense outside the context of RESTful web services. But you don’t have any use for links here, so you’ll go with plain-vanilla DTOs. Spring HATEOAS may be useful for implementing message-bus CDMs in addition to REST APIs.

app. In effect, you can use SI to create app-specific adapters to the RabbitMQ messaging infrastructure. Review figure 13.7 for a visual.

This section focuses on implementing the portal app’s outbound messages; that is, you’ll implement the TicketGateway interface. To complete the circuit, you also need to handle inbound messages into the help desk, so you’ll do that as well.

We won’t cover requests originating from the help desk app because the logic involved is more of the same. Refer to the sample code if you want to see it.

Let’s start by implementing the integration logic for the portal’s self-service ticket creation feature.

**IMPLEMENTING SELF-SERVICE TICKET CREATION: PORTAL’S OUTBOUND MESSAGING**

TicketGateway has a createTicket(Ticket) method that serves as a nice starting point because it’s fairly straightforward. The idea is that the customer creates a ticket using the portal’s web interface, and the portal passes it along to TicketGateway. Behind the scenes, the gateway does an asynchronous fire-and-forget at the messaging infrastructure, meaning that the call returns immediately. Later we’ll look at the message-handling code on the help desk side, but to keep things simple let’s focus on the portal’s fire-and-forget code.

Figure 13.8 shows what this looks like using the EIP graphical language. Note that the Spring Tool Suite generates these diagrams automatically from the SI configuration files; click the Integration-Graph tab in the configuration file editor.

The pipeline is straightforward. At the front end is a TicketGateway proxy that accepts requests from the application through the TicketGateway interface. It passes ticket creation requests to the AMQP channel adapter by way of a channel, and the channel adapter in turn pushes the message to a RabbitMQ exchange. In the case of ticket creation, all of this is completely asynchronous, so control returns to the portal immediately after invoking the TicketGateway. The following listing shows how to implement the pipeline using SI, Spring Rabbit, and Spring AMQP.

### Listing 13.11 beans-integration.xml: portal application

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:int-context="http://www.springframework.org/schema/context"
    xmlns:int-amqp="http://www.springframework.org/schema/integration/amqp"
```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns:oxm="http://www.springframework.org/schema/oxm"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:rabbit="http://www.springframework.org/schema/rabbit"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/
    spring-context-3.1.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/
    spring-integration-2.2.xsd
http://www.springframework.org/schema/integration/amqp
http://www.springframework.org/schema/integration/amqp/
    spring-integration-amqp-2.2.xsd
http://www.springframework.org/schema/oxm
http://www.springframework.org/schema/oxm/spring-oxm-3.1.xsd
http://www.springframework.org/schema/rabbit
http://www.springframework.org/schema/rabbit/spring-rabbit-1.1.xsd">
    <context:property-placeholder
        location="classpath:/spring/environment.properties" />
    <rabbit:connection-factory id="rabbitConnectionFactory"
        username="${rabbitMq.username}"
        password="${rabbitMq.password}" />
    <rabbit:admin
        connection-factory="rabbitConnectionFactory" />
    <rabbit:queue name="createTicketRequest.queue" />
    <rabbit:template id="amqpTemplate"
        connection-factory="rabbitConnectionFactory"
        message-converter="marshallingMessageConverter" />
    <bean id="marshallingMessageConverter"
        class="org.springframework.amqp.support.converter.MarshallingMessageConverter"
        p:contentType="application/xml">
        <constructor-arg ref="marshaller" />
    </bean>
    <oxm:jaxb2-marshaller id="marshaller">
        <oxm:class-to-be-bound
            name="com.springinpractice.ch13.cdm.Ticket" />
        <oxm:class-to-be-bound
            name="com.springinpractice.ch13.cdm.TicketStatus" />
        <oxm:class-to-be-bound
            name="com.springinpractice.ch13.cdm.TicketCategory" />
    </oxm:jaxb2-marshaller>
    <int:gateway
            gateway.TicketGateway"
        default-request-timeout="2000">
Implementing a message bus using RabbitMQ and Spring Integration

```
<int:method name="createTicket"
    request-channel="createTicketRequestChannel" />
</int:gateway>
<int:channel id="createTicketRequestChannel" />
<int-amqp:outbound-channel-adapter
    amqp-template="amqpTemplate"
    channel="createTicketRequestChannel"
    routing-key="createTicketRequest.queue" />
</beans>
```

Quite a bit is happening in this listing, but you can break the configuration into three sections: RabbitMQ, Object/XML mapping (OXM), and SI.

First, the RabbitMQ configuration begins with a connection factory. (Note that the default credentials for a fresh RabbitMQ installation are guest/guest.) You use `<rabbit:admin/>` at C to create queues dynamically if they don’t already exist. At 3 you declare a single queue for ticket-creation requests.

At 4 you create a template for sending messages to Rabbit. This follows Spring’s general practice of template-based communication with external systems and resources. The template uses a MarshallingMessageConverter (part of Spring AMQP) at 5 to perform OXM on message payloads. By default, the Rabbit template uses a SimpleMessageConverter, which handles strings, Serializable instances, and byte arrays. Because you want an XML-based CDM, you need a converter that performs OXM.

You configure a JAXB marshaller at 6, declaring the Ticket, TicketCategory, and TicketStatus DTOs for OXM binding. The MarshallingMessageConverter uses this marshaller.

The rest of the configuration is for SI. At 7 you define a dynamic proxy for the TicketGateway interface. The configuration at 8 routes tickets coming in through the createTicket() method to the createTicketRequestChannel 9, where the AMQP outbound channel adapter 10 receives it and pushes it to Rabbit’s default exchange, because you haven’t specified an exchange explicitly. This channel adapter, like all channel adapters, is unidirectional. (Gateways support bidirectional, request/reply messaging, but you don’t require that here.) The channel adapter’s routing key is set to createTicketRequest.queue, so the default exchange routes it to that queue. The message payload is ticket XML in the canonical format because the adapter uses the Rabbit template, which in turn uses the MarshallingMessageConverter.

That takes care of the fire-and-forget implementation of ticket creation on the portal side. Now there’s a message with an XML ticket payload sitting in a request queue on your bus. The next step is to implement integration logic on the help desk side to receive and service the request.

**IMPLEMENTING SELF-SERVICE TICKET CREATION: HELP DESK’S INBOUND MESSAGING**

This section shows how to process inbound ticket-creation requests. See figure 13.9 for a diagram showing how this works.

Here’s what’s happening. An inbound channel adapter receives the request from Rabbit, maps the ticket XML to a ticket DTO, and passes it to a processing chain. The
chain is a wrapper around a linear sequence of endpoints, obviating the need to define explicit channels connecting the chain’s members. The chain’s first endpoint is a transformer (SI’s terminology for EIP’s message translator) that maps the DTO to a ticket entity. Then a service activator invokes the TicketRepository.save(TicketEntity) method to save the ticket to the database. The repository’s save() method returns the saved instance, but the chain discards that message by dropping it onto the global nullChannel, which is essentially a black hole like /dev/null in Unix. Here’s the configuration for the integration logic just described.

**Listing 13.12 beans-integration.xml: help desk application**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:int="http://www.springframework.org/schema/integration"
    xmlns:int-amqp="http://www.springframework.org/schema/integration/amqp"
    xmlns:int-xml="http://www.springframework.org/schema/integration/xml"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:rabbit="http://www.springframework.org/schema/rabbit"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/integration
        http://www.springframework.org/schema/integration/spring-integration-2.2.xsd"
    ...>
```

Figure 13.9 A help desk integration pipeline that receives ticket-creation messages from the bus and creates tickets in the help desk database
Implementing a message bus using RabbitMQ and Spring Integration

http://www.springframework.org/schema/integration/amqp
http://www.springframework.org/schema/integration/amqp/
  spring-integration-amqp-2.2.xsd
http://www.springframework.org/schema/integration/xml
http://www.springframework.org/schema/integration/xml/
  spring-integration-xml-2.2.xsd
http://www.springframework.org/schema/oxm
http://www.springframework.org/schema/oxm/spring-oxm-3.1.xsd
http://www.springframework.org/schema/rabbit
http://www.springframework.org/schema/rabbit/spring-rabbit-1.1.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.1.xsd

```xml
<context:property-placeholder
  location="classpath:/spring/environment.properties" />

<rabbit:connection-factory id="rabbitConnectionFactory"
  username="${rabbitMq.username}"
  password="${rabbitMq.password}" />

<rabbit:admin connection-factory="rabbitConnectionFactory" />

<rabbit:queue name="createTicketRequest.queue" />

<bean id="marshallingMessageConverter
  class="org.springframework.amqp.support.converter.
  MarshallingMessageConverter"
  p:contentType="application/xml">
  <constructor-arg ref="marshaller" />
</bean>

<oxm:jaxb2-marshaller id="marshaller">
  <oxm:class-to-be-bound
    name="com.springinpractice.ch13.cdm.Ticket" />
  <oxm:class-to-be-bound
    name="com.springinpractice.ch13.cdm.TicketCategory" />
  <oxm:class-to-be-bound
    name="com.springinpractice.ch13.cdm.TicketStatus" />
</oxm:jaxb2-marshaller>

  helpdesk.integration.transformer" />

<int-amqp:inbound-channel-adapter
  queue-names="createTicketRequest.queue"
  channel="createTicketRequestChannel"
  message-converter="marshallingMessageConverter" />

<int:channel id="createTicketRequestChannel" />
<int:chain input-channel="createTicketRequestChannel"
  output-channel="nullChannel">
  <int:transformer ref="ticketTransformer" method="toEntity" />
  <int:service-activator
    expression="@ticketRepository.save(payload)" />
</int:chain>
</beans>
```
As with the portal application, you have an initial RabbitMQ configuration 1, although this time you don’t need a template. You also have the OXM configuration 2. This time around you have some transformers (more on that in a minute), so you scan for them at 3.

In listing 13.11 you had an AMQP outbound channel adapter to send messages to the bus, so here you have the inbound counterpart 4. The inbound channel adapter receives ticket-creation requests from createTicketRequest.queue and passes them via a channel 5 to a chain 6.

A chain is a linear sequence of endpoints connected by implicit channels. The first endpoint is a transformer 7 that transforms the ticket DTO into a ticket entity, as you’ll see. The second endpoint is a service activator 8 that saves the ticket entity to the Spring Data JPA ticket repository using a Spring Expression Language (SpEL) expression. The variables headers and payload are available for use, although you’re using only payload here. The payload is the ticket entity that the transformer generated. The call to save() returns the saved entity, but you don’t want to return that to the original caller; you send it to the global nullChannel 6, which sends the message to a black hole.

Next is the transformer that converts ticket DTOs into ticket entities.

Listing 13.13 TicketTransformer.java

```java
package com.springinpractice.ch13.helpdesk.integration.transformer;
import javax.inject.Inject;
import org.springframework.stereotype.Component;
import com.springinpractice.ch13.cdm.Ticket;
import com.springinpractice.ch13.helpdesk.model.TicketEntity;

@Component
public class TicketTransformer {
    @Inject private TicketCategoryTransformer ticketCategoryTransformer;
    @Inject private TicketStatusTransformer ticketStatusTransformer;

    public TicketEntity toEntity(Ticket ticketDto) {
        TicketEntity ticketEntity = new TicketEntity();
        ticketEntity.setCategory(
            ticketCategoryTransformer.toEntity(ticketDto.getCategory()));
        Customer customerDto = ticketDto.getCreatedBy();
        String username = customerDto.getUsername();
        if (username != null) {
            ticketEntity.setCustomerUsername(username);
        } else {
            ticketEntity.setCustomerEmail(customerDto.getEmail());
            ticketEntity.setCustomerFullName(getFullName(customerDto));
        }
        ticketEntity.setDateCreated(ticketDto.getDateCreated());
        ticketEntity.setDescription(ticketDto.getDescription());
        ticketEntity.setStatus( 
            ticketStatusTransformer.toEntity(ticketDto.getStatus()));
        return ticketEntity;
    }
}
```
```java
private String getFirstName(Customer customerDto) {
    String firstName = customerDto.getFirstName();
    String lastName = customerDto.getLastName();
    if (firstName == null) {
        return (lastName == null ? "[Unknown]" : lastName).trim();
    } else {
        return (lastName == null ? firstName : firstName + " " + lastName).trim();
    }
}

private String getFullName(Customer customerDto) {
    String firstName = customerDto.getFirstName();
    String lastName = customerDto.getLastName();
    if (firstName == null) {
        return (lastName == null ? "[Unknown]" : lastName).trim();
    } else {
        return (lastName == null ? firstName : firstName + " " + lastName).trim();
    }
}
```

The transformer is a POJO. Although it’s possible to use annotations to configure SI components, you’re using XML because I (Willie) find it easier to understand when the SI configuration is in one place.

At 1 you have a transformer method. This is the toEntity() method specified in listing 13.12. When there’s a single public method, you don’t have to specify the transformer method explicitly in the XML, but you do it anyway. Because the ticket DTO has references to category and status DTOs, you delegate the transformation to corresponding transformers 2 and 3.

With that, you have a full asynchronous flow from the portal application through the message bus and ending with the help desk. To be sure, there are some details we’ve neglected, such as error handling. But the basic integration is in place. The next section looks at a more complex case: implementing synchronous finder methods.

**IMPLEMENTING THE FINDERS: PORTAL’S OUTBOUND MESSAGING**

Finder methods involve a request/reply communication style, which takes more effort to implement in a messaging environment than the fire-and-forget style does. In this case you’ll implement synchronous request/replies, meaning the caller will block until the reply arrives; but note that SI also supports asynchronous request/replies, which are based on a callback mechanism. We won’t cover that here, though.

---

**Integration and services: an architectural perspective**

You might fairly ask why you would implement synchronous request/reply on top of a fundamentally asynchronous messaging infrastructure. Wouldn’t it be simpler to have the caller invoke a web service on the target system?

In many cases it’s indeed simpler to make a web service call. You can avoid implementing a bunch of integration patterns on the bus, as well as avoid forcing the request and reply messages to pass through the message bus.

But the arguments for using a bus for asynchronous communications mostly apply even for synchronous communications: (1) client systems can decouple themselves from service-specific locations, message formats, authentication schemes, and so on; and (2) you avoid the aforementioned $O(n^2)$ problem associated with point-to-point messaging.
One pro-bus argument that doesn’t apply in the case of synchronous messaging is the runtime decoupling argument. In the asynchronous case, it doesn’t matter if a message receiver is offline when the sender sends the message, because the messaging system queues the message until the receiver is available. With synchronous communications, the receiver must be available when the sender sends it a request.

We won’t settle the issue here, but suffice it to say there’s a design decision to consider. The rest of the recipe shows how to implement synchronous messaging without necessarily claiming that it’s the right approach for all cases.

You’ll add support for three finder methods. Figure 13.10 augments the portal-side pipeline you established in figure 13.8. Originally you had a single path to an AMQP outbound channel adapter. This time you add a couple of new paths to an AMQP outbound gateway.

Channel adapters and gateways are alike in that they’re both interfaces to external systems, but not alike in that channel adapters are unidirectional (fire-and-forget) while gateways support request/reply communications. In this case, the external system is the message bus. The following listing shows how to implement the pipeline in figure 13.10.

**Listing 13.14 beans-integration.xml: portal application**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ... RabbitMQ configuration from listing 13.11, plus the following ...
  <rabbit:queue name="findTicketStatusRequest.queue" />
  <rabbit:queue name="findTicketCategoriesRequest.queue" />
  <rabbit:queue name="findTicketCategoryRequest.queue" />
</beans>
```
Implementing a message bus using RabbitMQ and Spring Integration

... OXM configuration from listing 13.11, plus the following ...

```
<oxm:jaxb2-marshaller id="marshaller">
  <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.DummyPayload" />
  <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.Ticket" />
  <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketCategory" />
  <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketCategoryRequest" />
  <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketStatus" />
  <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketStatusRequest" />
</oxm:jaxb2-marshaller>

<int:gateway service-interface="com.springinpractice.ch13.portal.integration.TicketGateway">
  default-request-channel="helpDeskRequestChannel"
  default-request-timeout="2000"
  default-reply-timeout="2000">
    <int:method name="createTicket" request-channel="createTicketRequestChannel" />
    <int:method name="findOpenTicketStatus" payload-expression="new com.springinpractice.ch13.cdm.TicketStatusRequest('open')">
      <int:header name="requestType" value="findTicketStatusRequest" />
    </int:method>
    <int:method name="findTicketCategories" payload-expression="new com.springinpractice.ch13.cdm.DummyPayload()">
      <int:header name="requestType" value="findTicketCategoriesRequest" />
    </int:method>
    <int:method name="findTicketCategory" request-channel="findTicketCategoryRequestChannel">
      <int:header name="requestType" value="findTicketCategoryRequest" />
    </int:method>
  </int:gateway>
  <int:channel id="createTicketRequestChannel" />
  <int:amqp:outbound-channel-adapter amqp-template="amqpTemplate" channel="createTicketRequestChannel" routing-key="createTicketRequest.queue" />
  <int:channel id="findTicketCategoryRequestChannel" />
  <int:transformer expression-based transformer="true" />
```

Additional DTOs for binding

Gateway dynamic proxy

Per-method enrichment and routing

5 Per-method enrichment and routing

6 Per-method enrichment and routing

7 Channel

8 Expression-based transformer
You add three new queues at ① to support your new finder methods. At ② you add more classes to be bound to the OXM configuration. You’ll see why you’re adding the dummy payload and special request objects in a moment.

You modify the gateway definition at ③. You specify that by default all requests coming into the gateway will land on the helpDeskRequestChannel. You also set a default reply timeout, expressed in milliseconds, because now you’re expecting replies.

On replies: unless you specify an explicit default-reply-channel (which you’re not doing here), the gateway creates for any given request a temporary, anonymous reply channel, and adds the channel to the request message as a header called replyChannel. That way, reply-generating downstream endpoints know where to place the reply.

The first finder method is findOpenTicketStatus() ④. You use a Sp EL payload expression to create a TicketStatusRequest DTO for the open status. The reason you create a special request DTO is that you need the request to be XML. This is because the AMQP gateway expects an XML reply from the bus (recall your CDM), which it maps to an object via the AMQP template, which in turn uses the MarshallingMessageConverter. The template applies the converter to both the request and the reply, so the request needs to be a mappable DTO as opposed to a simple string.

In addition to the Sp EL payload, the finder method definition includes a custom requestType header (custom in the sense that you invented it). Both SI and RabbitMQ support message headers, but here, the header is an SI header. You’ll use this header to route finder requests to the right queue, as you’ll see.

At ⑤ is the second finder method. This time you get a list containing all ticket categories, which is useful for populating the category drop-down in the new ticket form. There is a small problem, though. By default, SI treats no-arg gateway methods as connecting to pollable (receive-only) channels, as opposed to no-arg request/reply (send-then-receive) channels. To implement a request/reply communication, you need to provide a dummy payload using payload-expression. Normally you can pass in a dummy string or a Date:

payload-expression="new java.util.Date()"

But here that doesn’t work because you’re using MarshallingMessageConverter, which expects payloads to be mappable XML. That’s why you have the DummyPayload class, and you use payload-expression to create an instance here. Once again you enrich your message with a requestType header for routing purposes.
The third finder retrieves a specific ticket category by ID 6. Once again you need to represent the payload ID using XML rather than a Long. You’ll need a transformer for this. You override the gateway’s default request channel with a new channel called findTicketCategoryRequestChannel and then pass the message over that channel 7 to the transformer at 8. Here you take advantage of the transformer’s expression attribute to wrap the Long ID in a mappable request DTO. Finally the message goes to the helpDeskRequestChannel 9 like the other help desk requests.

The next stop is an AMQP outbound gateway 10. You use the AMQP template to do the actual request and reply. Then there are a couple of header-related attributes. First you use routing-key-expression to specify a dynamic, message-driven routing key that allows Rabbit’s default exchange to route messages to queues. In this case, the expression is a SpEL expression that appends .queue to the value of the requestType SI header you’ve been using.

You use mapped-request-headers to indicate that you want the SI requestType header to appear as an AQMP header as well once the message hits the bus. This is because you’ll have further use for this header for routing on the help desk side.

As with the initial gateway, the AMQP outbound gateway generates a reply. Here’s how this works behind the scenes. For any given request, the outbound gateway creates a temporary reply queue and sets the AMQP message’s reply_to property to the queue’s name. This tells downstream endpoints where to place the reply when it materializes. Once the reply appears in that queue, the AMQP outbound gateway grabs it and places it on the request message’s reply channel. You’ll recall from our discussion that the request message maintains a reference to the reply channel as the value of its replyChannel header.

IMPLEMENTING THE FINDERS: HELP DESK’S INBOUND MESSAGING

Now the portal sends finder requests to the bus, so the help desk needs to pick those up and service them. The supporting help desk pipeline appears in figure 13.11.

![Figure 13.11](image_url)  
**Figure 13.11**  The help desk’s inbound pipeline to support the TicketGateway’s finder methods. Although it’s not shown here, each chain contains a service activator followed by a transformer.
This help desk pipeline receives finder requests at an AMQP inbound gateway and forwards them to a router, which uses the requestType header to pass the request to one of three chains. Each chain invokes a finder method on the TicketRepository and uses a transformer to convert the result into a DTO before returning it to the caller. Here's the configuration you use to implement the help desk pipeline.

Listing 13.15  beans-integration.xml: help desk application

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    ... RabbitMQ configuration from listing 13.12, plus the following ...

    <rabbit:queue name="findTicketStatusRequest.queue" />
    <rabbit:queue name="findTicketCategoriesRequest.queue" />
    <rabbit:queue name="findTicketCategoryRequest.queue" />

    ... OXM configuration from listing 13.12, plus the following ...

    <oxm:jaxb2-marshaller id="marshaller">
        <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.DummyPayload" />
        <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.Ticket" />
        <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketCategory" />
        <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketCategory$TicketCategoryList" />
        <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketCategoryRequest" />
        <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketStatus" />
        <oxm:class-to-be-bound name="com.springinpractice.ch13.cdm.TicketStatusRequest" />
    </oxm:jaxb2-marshaller>

    <context:component-scan base-package="com.springinpractice.ch13.helpdesk.integration.transformer" />

    ... inbound create ticket pipeline from listing 13.12 ...

    <int-amqp:inbound-gateway
        queue-names="findTicketStatusRequest.queue,
        findTicketCategoriesRequest.queue,
        findTicketCategoryRequest.queue"
        request-channel="helpDeskRequestChannel"
        mapped-request-headers="requestType"
        message-converter="marshallingMessageConverter" />

    <int:channel id="helpDeskRequestChannel" />

    <int:header-value-router input-channel="helpDeskRequestChannel"
        header-name="requestType">
        <int:mapping value="findTicketStatusRequest"
            channel="findTicketStatusRequestChannel" />
        <int:mapping value="findTicketCategoriesRequest"
            channel="findTicketCategoriesRequestChannel" />
    </int:header-value-router>
</beans>
```
As was true with the portal SI configuration, you declare the three queues for finder requests at 1 to ensure that they exist. You also declare the same set of DTOs for OXM at 2 because you’ll need to convert back and forth between the bus CDM and Java.

You need a few transformers to convert the entities you find into DTOs, so you scan for them at 3. The entry point for synchronous messages into the pipeline is the AMQP inbound gateway at 4. You specify its three feeder queues using the queue-names attribute. Just as you used mapped-request-headers in listing 13.14 to convert the custom SI requestType header into an AMQP header, you use it here to convert the AMQP header back into a custom SI requestType header.

The AMQP inbound gateway supports replies. When the gateway receives a message from a queue, it creates an anonymous reply channel and attaches it to the message using the replyChannel message header. Eventually some downstream component responsible for producing the reply will place the reply in that channel.

The gateway passes requests to a router that uses header values to drive routing 5. As you’ve guessed, you’re using the requestType header for that. The <mapping> elements provide the routing definitions.

Once the request leaves the router, it goes to one of three chains you’ve defined, corresponding to the three finder requests. First is a chain for the ticket status requests 6. The chain has an expression-driven service activator that unpacks the key from the request object (recall that you wrapped the key with a TicketStatusRequest in listing 13.14) and calls the findByKey() method on the TicketStatusRepository. The result is an entity, so you use a transformer to convert the entity back to a DTO for
CHAPTER 13 Enterprise integration

subsequent mapping to the XML-based CDM on the return trip. See the sample code
for the transformers, which are similar to the one from listing 13.13.

Because you haven’t specified an explicit output channel for the chain, the chain
sends the transformer’s output to the channel you’re storing under the replyChannel
header. The circuit is now complete: the help desk AMQP inbound gateway receives
the reply from the channel and sends it to the specified exchange and queue (as speci-
ified by the routing key). The portal AMQP outbound gateway receives the reply from
the queue and places it on the replyChannel. Finally the initial portal gateway
receives the reply and returns it to the caller. The chains at and are essentially
similar to the one at 

With that, you’re done. You now have the plumbing on both the portal and the
help desk sides to support both asynchronous and synchronous communications over
Rabbit. Although we didn’t cover it here, note that the help desk also requests cus-
tomer information from the portal, using largely the same set of patterns, but in the
opposite direction. See the sample code.

Discussion
Over the past three recipes we’ve shown how to integrate applications in a progres-
sively more decoupled way. Though we’ve considered only two apps here, the archi-
tecture’s power becomes more obvious as you place additional apps on the message
bus. The number of potential integrations grows quadratically in the number of apps,
but the integration complexity increases only linearly.

In this recipe you used RabbitMQ as the bus-implementation technology and SI as
a way to implement app-specific bus adapters. But this isn’t the only way to use SI. You
can use SI itself to implement buses. In the recipes that follow, you’ll reposition the
help desk’s SI pipeline as an application bus in its own right and then add both
inbound and outbound email by attaching them to the application bus.

13.4 Sourcing tickets from an IMAP store

PREREQUISITES
Recipe 13.3 Implementing a message bus using RabbitMQ and Spring Integration
(There’s no conceptual dependency on RabbitMQ, but you use the code from reci-
pe 13.3.)

KEY TECHNOLOGIES
SI, JavaMail, IMAP

Background
Email-based support is a common requirement. Although many sites offer a form-
based option to better structure the ticket and to avoid email spam, email can be an
attractive option because it’s so easy to implement: all it requires is an inbox.

In this recipe, imagine that you only recently rolled out the form-based approach
from recipe 13.1, but you still want to support a legacy support email address that was
your primary ticket source prior to introducing the form. We’ll show how to create help desk tickets based on incoming customer email.

**Problem**

Automatically create help desk tickets based on customer email.

**Solution**

Building on figure 13.7, figure 13.12 shows what you want to add to the integration landscape in this recipe.

One question you might be asking is why you wouldn’t attach inbound email to the RabbitMQ bus instead of attaching it to the help desk’s SI adapter. After all, the portal is a ticket source, and you’ve attached it to the RabbitMQ bus. And in the help desk, you’re using SI as an adapter to the RabbitMQ bus, so accepting email from a source other than the bus seems to conflict with this design.

You could certainly do that, but one reason you’re not is that you’d need a separate adapter to connect the inbound email channel to RabbitMQ, and that’s a complexity you don’t currently require. Only the help desk cares about inbound email, and if the help desk isn’t available to receive email, then messages sit in the mailbox until the help desk is available again. (In effect, the mailbox functions as a persistent message queue.)

As to the design conflict, the conflict is only apparent. All the help desk sees are the gateway interfaces you happen to have in place; the app doesn’t know anything about SI or RabbitMQ. Instead of thinking of the SI pipeline strictly as an adapter to the RabbitMQ bus, you can consider it to be an application bus in a federated bus

---

![Figure 13.12](image-url) You’ll add an email-based ticket channel using SI’s support for inbound email.
architecture, one that connects the app to external buses and systems. This includes RabbitMQ, but it can also include systems whose use is limited to specific applications, like inbound email in the current instance.

**Spring Integration supports multiple integration architectures**

The preceding discussion highlights the fact that SI is flexible; it doesn’t prescribe a specific integration architecture. You can have a single central message broker with SI adapters if you like. You can have federated, hierarchical buses. Or you can even use SI itself as a central bus.

Figure 13.13 presents graphically the pipeline you’re going to create. It builds on the pipeline from figure 13.9.

You’re adding an IMAP inbound channel adapter to receive email messages from an IMAP mailbox. Then you pass the email messages to a transformer, which converts them into ticket DTOs. (The DTOs are the canonical data model for the help desk’s application bus.) The transformer drops the DTOs onto the existing `createTicketRequestChannel`, which allows you to take advantage of the downstream chain for saving tickets you created in recipe 13.3, and also to use it as a single location for making changes to the integration logic. You’ll see this benefit in action when you add confirmation emails in recipe 13.5.

As before, you use SI to implement the pipeline.

**SPRING INTEGRATION CONFIGURATION**

Listing 13.16 shows what you need to add to your help desk `beans-integration.xml` configuration to support the pipeline depicted. *But first, please read the following warning.*
WARNING: listing 13.16 deletes all of your email!

The following configuration treats your IMAP mailbox as a message queue. The IMAP channel adapter treats every email in the mailbox as a message to be processed and deleted. **Please use a test email account, not your personal or work account.** I (Willie) learned this the hard way by stupidly deleting several years of Gmail messages from my personal inbox.

Now that you’ve read the warning and created a test account, please see the following listing to add support for inbound email.

**Listing 13.16  Help desk’s beans-integration.xml, with support for inbound email**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns:int-mail="http://www.springframework.org/schema/integration/mail"
    xmlns:property="http://www.springframework.org/schema/property"
    xmlns:transformer="http://www.springframework.org/schema/integration/transformer"
    xmlns:channel="http://www.springframework.org/schema/integration/channel"
    xmlns:poller="http://www.springframework.org/schema/integration/poller"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
    http://www.springframework.org/schema/integration/mail
    http://www.springframework.org/schema/integration/mail/
    spring-integration-mail-2.2.xsd
    http://www.springframework.org/schema/property
    http://www.springframework.org/schema/property/
    spring-integration-property-2.2.xsd
    http://www.springframework.org/schema/integration/transformer
    http://www.springframework.org/schema/integration/transformer/
    spring-integration-transformer-2.2.xsd
    http://www.springframework.org/schema/integration/channel
    http://www.springframework.org/schema/integration/channel/
    spring-integration-channel-2.2.xsd
    http://www.springframework.org/schema/integration/poller
    http://www.springframework.org/schema/integration/poller/
    spring-integration-poller-2.2.xsd"
>
    ... configuration from recipe 13.3 ...

    <int-mail:imap-idle-channel-adapter
        channel="newMailChannel"
        store-uri="${email.store.uri}"
        should-delete-messages="${email.shouldDeleteMessages}" />

    <int:channel id="newMailChannel" />

    <int:transformer input-channel="newMailChannel"
        output-channel="createTicketRequestChannel"
        expression="@ticketTransformer.toDto(payload)" />

</beans>
```

Little is involved here. You have the IMAP inbound channel adapter 1. This is a special IMAP IDLE adapter, which supports the IMAP IDLE notification mechanism. If your provider doesn’t support IMAP IDLE, then you can use a standard IMAP inbound channel adapter with a poller:

```xml
<int-mail:inbound-channel-adapter
    channel="newMailChannel"
    store-uri="${email.store.uri}"
    should-delete-messages="${email.shouldDeleteMessages}" />

<int:poller max-messages-per-poll="3" fixed-rate="30000" />
</int-mail:inbound-channel-adapter>
```

In any event, you specify the IMAP store (mailbox) URI and also tell the channel adapter to go ahead and delete messages from the mailbox after pulling them down.
Different services will have different URIs. For Gmail, it looks like this:

imaps://username:password@imap.gmail.com:993/Inbox

Notice the use of IMAPS, which is IMAP over SSL (standard port is 993).\(^{15}\) Obviously you need to replace `username` and `password` with the actual credentials associated with the account.

After the channel adapter receives an email message, it sends it to a transformer so that it can be converted into a DTO. You’re using the transformer’s expression attribute to select the transformation. After that it goes to the `createTicketRequestChannel`, where the chain from recipe 13.3 receives it and saves it to the `TicketRepository`.

The transformer code is important, so let’s look at that.

### UPDATING THE TICKETTRANSFORMER

The transformer is an updated version of `TicketTransformer` from listing 13.13. Here’s the new version.

**Listing 13.17 Updated version of TicketTransformer.java**

```java
package com.springinpractice.ch13.helpdesk.integration.transformer;
import java.io.IOException;
import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.mail.BodyPart;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;
import org.springframework.stereotype.Component;
import com.springinpractice.ch13.cdm.Ticket;
import com.springinpractice.ch13.cdm.TicketCategory;
import com.springinpractice.ch13.cdm.TicketStatus;
import com.springinpractice.ch13.helpdesk.model.TicketCategoryEntity;
import com.springinpractice.ch13.helpdesk.model.TicketEntity;
import com.springinpractice.ch13.helpdesk.model.TicketStatusEntity;
import com.springinpractice.ch13.helpdesk.repo.TicketCategoryRepository;
import com.springinpractice.ch13.helpdesk.repo.TicketStatusRepository;
@Component
public class TicketTransformer {
    @Inject private TicketCategoryRepository ticketCategoryRepo;
    @Inject private TicketStatusRepository ticketStatusRepo;
    @Inject private TicketCategoryTransformer ticketCategoryTransformer;
    @Inject private TicketStatusTransformer ticketStatusTransformer;
```

---

\(^{15}\) If you run into PKIX/certificate trust issues, you may need to import the Gmail IMAP certificate into your truststore. See Willie Wheeler, “Fixing PKIX path building issues when using JavaMail and SMTP,” Spring in Practice, [http://mng.bz/W4w8](http://mng.bz/W4w8). This discussion involves SMTP, but with minor modifications it applies to IMAP as well.
private TicketCategory generalCategoryDto;
private TicketStatus openStatusDto;

@PostConstruct
public void postConstruct() {
    TicketCategoryEntity generalCategoryEntity =
        ticketCategoryRepo.findByKey("general");
    this.generalCategoryDto =
        ticketCategoryTransformer.toDto(generalCategoryEntity);
    TicketStatusEntity openStatusEntity =
        ticketStatusRepo.findByKey("open");
    this.openStatusDto =
        ticketStatusTransformer.toDto(openStatusEntity);
}

public TicketEntity toEntity(Ticket ticketDto) {
    ... same as listing 13.13 ...
}

public Ticket toDto(MimeMessage email)
    throws MessagingException, IOException {
    InternetAddress from = (InternetAddress) email.getFrom()[0];
    MimeMultipart content = (MimeMultipart) email.getContent();
    BodyPart body = content.getBodyPart(0);
    Ticket ticketDto = new Ticket();
    ticketDto.setCategory(generalCategoryDto);
    Customer customerDto = new Customer();
    customerDto.setEmail(from.getAddress());
    customerDto.setFirstName(null);
    customerDto.setLastName(from.getPersonal());
    ticketDto.setCreatedBy(customerDto);
    ticketDto.setDateCreated(email.getSentDate());
    ticketDto.setDescription("[" + email.getSubject() + "] " + body.getContent());
    ticketDto.setStatus(openStatusDto);
    return ticketDto;
}

    ... getFullName() same as listing 13.13 ...
}

You use the @PostConstruct annotation 1 to declare a method for Spring to run
after creating and injecting the bean. You use this to preload the General ticket cate-
gory (the support rep can change it to something more appropriate) and the Open
ticket status.

The actual transformation occurs at 2. The IMAP channel adapter produces a
MimeMessage, so you transform that into a DTO so the downstream chain can save it.

That’s it for the code and configuration. It’s time to try it out.

TRY THE CODE
Choose a test account for your IMAP store, and, if you’re using GitHub code, change
should-delete-messages from false to true on the IMAP inbound channel adapter.
Then start up the help desk app and send an email to the test account. The channel adapter should see the email, grab it, delete it from the mailbox, and then turn it into a ticket. You can see the ticket by viewing the ticket list in the help desk’s UI.

Discussion

In this recipe, you learned that it’s easy to add support for inbound email to a Spring-enabled application. This is useful because email is still a popular way to allow users to submit support requests and other communications.

In the following recipe, we’ll revisit the topic of confirmation emails, which you saw in chapter 8. This time you’ll use SI to send the confirmation email.

13.5 Send confirmation messages over SMTP

PREREQUISITES
Recipe 13.4 Sourcing tickets from an IMAP store

KEY TECHNOLOGIES
SI, JavaMail, SMTP

Background

Generally, when users submit support tickets, you want to send them a confirmation message thanking them for their ticket and letting them know when they can expect to hear back from you.

Problem

Send the user a confirmation email when they submit a ticket.

Solution

Figure 13.14 shows the last step in the evolution of your integration environment. This time you’re adding support for confirmation emails, which you send by way of SMTP. This is the same as figure 13.1, but it’s reproduced here for your convenience.

As it happens, you can add confirmation emails without changing any app code. Recall from recipe 13.4 that you connected the IMAP inbound channel adapter to a chain that your AMQP inbound channel adapter was already using for creating tickets. Because they’re both using the same pipeline, you can modify that pipeline a bit to
generate confirmation emails, regardless of whether the ticket came from the web form or an email. Figure 13.15 shows how. The following listing shows the required configuration updates.

**Listing 13.18  Help desk’s beans-integration.xml, with support for confirmation emails**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:jee="http://www.springframework.org/schema/jee"
 ... other namespaces ...
 xsi:schemaLocation=" ... other schema locations ... ">
 ... configuration from recipe 13.4 ...

<jee:jndi-lookup id="mailSession"
 jndi-name="mail/Sip13HelpDeskMailSession"
 resource-ref="true" />

<bean id="mailSender"
 class="org.springframework.mail.javamail.JavaMailSenderImpl"
 p:session-ref="mailSession" />

<int:publish-subscribe-channel id="createTicketRequestChannel" />

<int:transformer input-channel="createTicketRequestChannel"
 output-channel="confirmationEmailChannel"
 expression="@ticketTransformer.toConfirmationEmail(payload)" />

<int:channel id="confirmationEmailChannel" />

<int-mail:outbound-channel-adapter 
 channel="confirmationEmailChannel" mail-sender="mailSender" />
</beans>
```

First you have some JavaMail configuration 1. This is the same as what you saw in recipe 8.2.16

As before, if you run into PKIX issues, see “Fixing PKIX path building issues when using JavaMail and SMTP,” http://mng.bz/W4w8.
You replace the original point-to-point channel with a publish/subscribe (pub/sub) channel. The difference between them is that a point-to-point channel can have at most one consumer, whereas a pub/sub channel broadcasts messages to any number of consumers. Here you want to continue broadcasting to the chain that saves the ticket, but you want to add a new consumer pipeline to generate confirmation emails.

The start of that new pipeline is the transformer at \textcircled{3}. It converts ticket DTOs into outbound confirmation emails. Then you pass the emails along to an SMTP outbound channel adapter, which sends the email.

Next you update TicketTransformer to support confirmation emails.

### Listing 13.19 TicketTransformer.java with a transform method for confirmation emails

```java
package com.springinpractice.ch13.helpdesk.integration.transformer;
...
import org.springframework.beans.factory.annotation.Value;
import org.springframework.mail.MailMessage;
import org.springframework.mail.SimpleMailMessage;
@Component
public class TicketTransformer {
    ...
    @Value("${confirmation.from}")
    private String confirmationFrom;
    @Value("${confirmation.subject}")
    private String confirmationSubject;
    ...
    public MailMessage toConfirmationEmail(Ticket ticketDto) {
        MailMessage msg = new SimpleMailMessage();
        Customer customerDto = ticketDto.getCreatedBy();
        String customerFullName = getFullName(customerDto);
        String customerEmail = customerDto.getEmail();
        String to = (customerFullName == null ? customerEmail :
                  customerFullName + " <" + customerEmail + ">");
        msg.setTo(to);
        msg.setFrom(confirmationFrom);
        msg.setSubject(confirmationSubject);
        msg.setSentDate(new Date());
        String desc = "Thank you for reporting this issue. We will contact you " +
                      "within one business day. \n\nYour message: \n\n" +
                      ticketDto.getDescription();
        msg.setText(desc);
        return msg;
    }
}
```
You use `@Value` to inject a couple of confirmation email parameters into the transformer at 1. The new transform method creates a confirmation email from a ticket DTO 2. You create the email 3 and then use the DTO to populate its fields 4. With respect to the description 5, you hardcode the confirmation message, but in a more realistic example you would use a template engine (Velocity, FreeMarker, and so on) as you did in recipe 8.2.

Start up the help desk and the portal, and try creating messages through the portal, help desk, and email interfaces. In each case you should see the help desk generating confirmation emails. Note that you'll need to change the email addresses of the sample portal users to your own email address if you want to receive confirmation emails when submitting tickets involving those users.

**Discussion**

This recipe demonstrated that it’s possible to perform integrations without having to modify the apps. You added confirmation emails by replacing a point-to-point channel for ticket creation with a pub/sub channel and then attaching both the help desk service and the confirmation email pipeline to that channel.

Where you control the apps being integrated, it makes sense to consider combining integration logic with app modifications to eliminate redundancy and simplify integration. But this isn’t always possible. In general, this means you’ll want to create abstract representations of key actions on the bus. For example, before the integration, the “create ticket” action lived with the help desk. But to add a confirmation email, you had to represent that action in the bus and treat the actual ticket creation as just one flow out of the bus. Ultimately, the services become implementation details to the logical representations on the bus, which makes the architecture and services easier to evolve over time.

**13.6 Summary**

Integration is an important concern in enterprise environments, where there is generally a bewildering array of both complementary and competing tools in place, often with little hope of long-term harmonization. Integration becomes that harmonization—it provides a practical way to connect tools and their data to support higher-level process and workflow integration.

There are many approaches to integrating systems, and we’ve covered some of the important ones here. When custom, internally developed code is involved, shared databases can offer a simple and quick way to make data broadly available. But this approach scales poorly, and so the next step is often to use web services to enhance decoupling. Finally, domain- or even enterprise-level, broker-based messaging is a powerful way to increase decoupling even further, which becomes important as the number of collaborators in the integration grows.

Spring provides a number of APIs useful for integration styles, including Spring Data REST, Spring HATEOAS, Spring Integration, and Spring AMQP/Rabbit. Integration is
such a large topic that it’s impossible for a single chapter to do more than touch on the complexities and solutions involved, but we’ve tried to offer a starting point to support further exploration and study. *Spring Integration in Action* and *RabbitMQ in Action*, both published by Manning, are great places to start.

In the next and final chapter of the book, you’ll learn how to use Spring to create your own framework, complete with annotations and namespace configuration.
Spring in Practice covers 66 Spring development techniques and the practical issues you will encounter when using them. The book starts with three carefully crafted introductory chapters to get you up to speed on the fundamentals. And then, the core of the book takes you step by step through the important, practical techniques you will use no matter what type of application you’re building. You’ll hone your Spring skills with examples on user accounts, security, NoSQL data stores, and application integration. Along the way, you’ll explore Spring based approaches to domain-specific challenges like CRM, configuration management, and site reliability.

What’s Inside
- Covers Spring 3
- Successful outcomes with integration testing
- Dozens of web app techniques using Spring MVC
- Practical examples and real-world context
- How to work effectively with data

Each technique highlights something new or interesting about Spring and focuses on that concept in detail. This book assumes you have a good foundation in Java and Java EE. Prior exposure to Spring Framework is helpful but not required.

Willie Wheeler is a Principal Applications Engineer with 16 years of experience in Java/Java EE and Spring Framework. Joshua White is a Solutions Architect in the financial and health services industries. He has worked with Spring Framework since 2002.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SpringinPractice