Extracted from:

Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

The Pragmatic Bookshelf

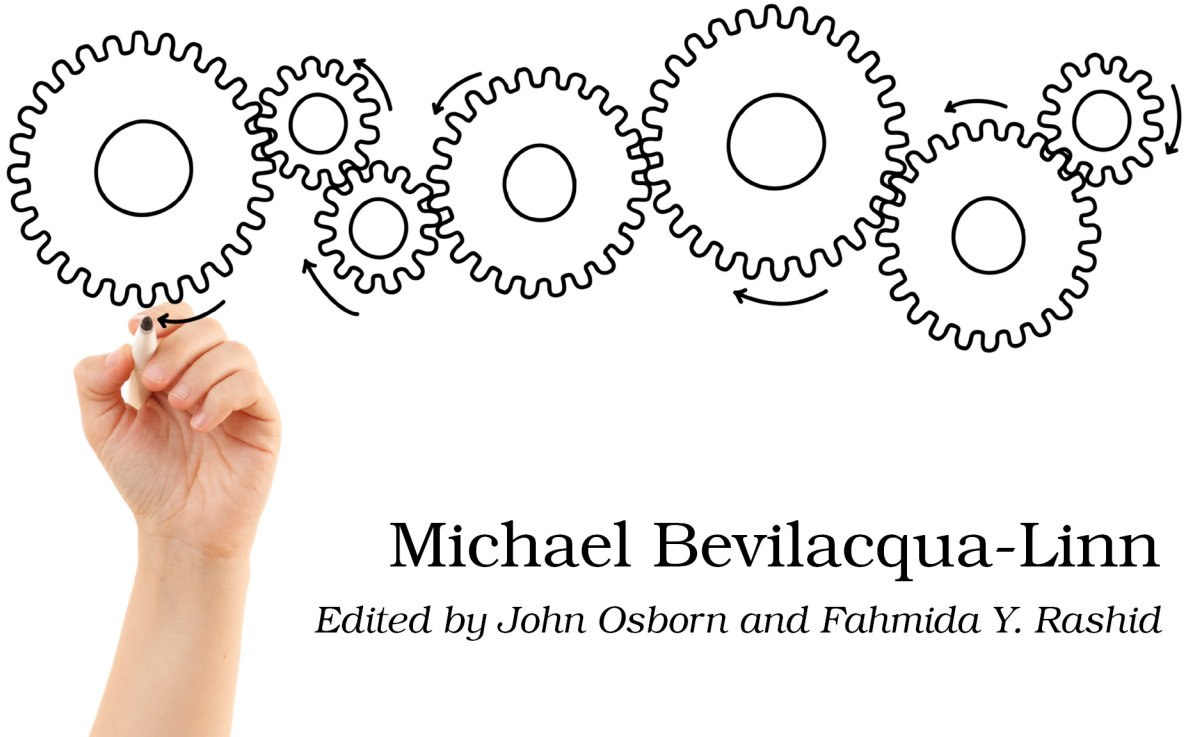
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming Patterns

in Scala and Clojure

Write Lean Programs for the JVM



Michael Bevilacqua-Linn

Edited by John Osborn and Fahmida Y. Rashid

Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

Michael Bevilacqua-Linn

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

2.1 Introducing TinyWeb

We'll start our journey with a look at an example of a program that makes heavy use of classic object-oriented patterns, a small web framework called TinyWeb. After introducing TinyWeb, we'll see how to rewrite it in a hybrid object-oriented and functional style using Scala. Finally, we'll move on to a more fully functional style in Clojure.

Let's focus on a few goals for this example. The first is to see several patterns working together in one codebase before we go into them in more detail.

The second is to introduce basic Scala and Clojure concepts for those unfamiliar with either, or both, of the languages. A full introduction to the languages is beyond the scope of this book, but this section gives you enough of the basics to understand the majority of the remaining code.

Finally, we'll work existing Java code into a Scala or Clojure codebase. We'll do this by taking the Java version of TinyWeb and transforming it into Scala and Clojure piece by piece.

TinyWeb itself is a small *model-view-controller* (MVC) web framework. It's far from complete, but it should feel familiar to anyone who has worked with any of the popular frameworks, such as Spring MVC. There's one little twist to TinyWeb: since this is a book on functional programming, we're going to do our best to work with immutable data, which can be quite challenging in Java.

2.2 TinyWeb in Java

The Java version of TinyWeb is a basic MVC web framework written in a classic object-oriented style. To handle requests we use a `Controller` implemented using the Template method, which we cover in detail in Pattern 6, *Replacing Template Method*, on page ?. Views are implemented using the Strategy pattern, covered in Pattern 7, *Replacing Strategy*, on page ?.

Our framework is built around core pieces of data objects, `HttpRequest` and `HttpResponse`. We want these to be immutable and easy to work with, so we are going to build them using the Builder pattern discussed in Pattern 4, *Replacing Builder for Immutable Object*, on page ?. Builder is a standard way of getting immutable objects in Java.

Finally, we've got request filters that run before a request is handled and that do some work on the request, such as modifying it. We will implement these filters using the `Filter` class, a simple example of Pattern 1, *Replacing Functional*

Interface, on page ?. Our filters also show how to handle changing data using immutable objects.

The whole system is summarized in the following figure.

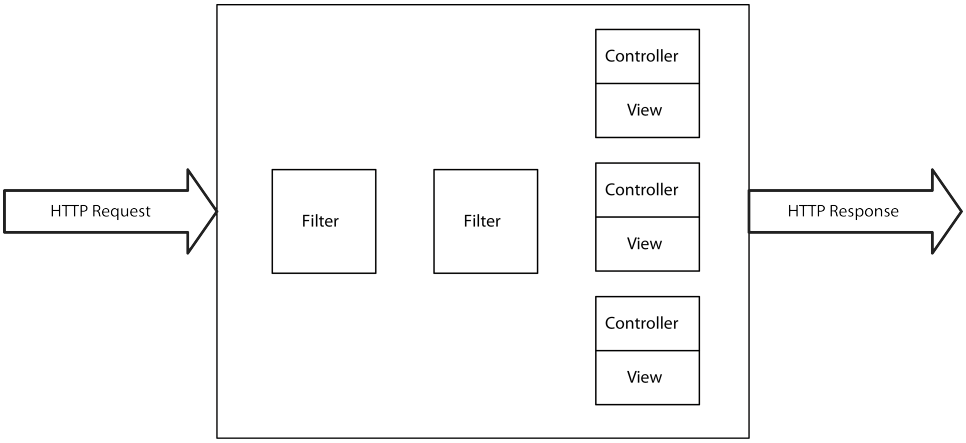


Figure 1—A TinyWeb Overview. A graphical overview of TinyWeb

We'll start off with a look at our core data types, `HttpRequest` and `HttpResponse`.

HttpRequest and HttpResponse

Let's dig into the code, starting with `HttpResponse`. In this example we'll only need a body and a response code in our response, so those are the only attributes we'll add. The following code block shows how we can implement the class. Here we use the fluent builder of the type made popular in the Java classic, *Effective Java [Blo08]*.

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/HttpResponse.java

```
package com.mblinn.oo.tinyweb;

public class HttpResponse {
        private final String body;
        private final Integer responseCode;

        public String getBody() {
                return body;
        }

        public Integer getResponseCode() {
                return responseCode;
        }
```

```
        private HttpResponse(Builder builder) {
                body = builder.body;
                responseCode = builder.responseCode;
        }

        public static class Builder {
                private String body;
                private Integer responseCode;

                public Builder body(String body) {
                        this.body = body;
                        return this;
                }

                public Builder responseCode(Integer responseCode) {
                        this.responseCode = responseCode;
                        return this;
                }

                public HttpResponse build() {
                        return new HttpResponse(this);
                }

                public static Builder newBuilder() {
                        return new Builder();
                }
        }
}
```

This approach encapsulates all mutability inside of a `Builder` object, which then builds an immutable `HttpResponse`. While this gives us a clean way of working with immutable data, it's quite verbose. For example, we could create a simple test request using this code:

```
HttpResponse testResponse = HttpResponse.Builder.newBuilder()
        .responseCode(200)
        .body("responseBody")
        .build();
```

Without using Builder we'd need to pass all of our arguments in the constructor. This is okay for our small example, but this practice grows unwieldy when working with larger classes. Another option would be to use a Java Bean–style class with getters and setters, but that would require mutability.

Let's move on and take a quick look at HttpRequest. Since the class is similar to HttpResponse (though it lets us set a request body, headers, and a path), we won't repeat the code in full. One feature is worth mentioning, though.

In order to support request filters that "modify" the incoming request, we need to create a new request based off the existing one, since our request objects aren't mutable. We'll use builderFrom() to do so. This method takes an existing HttpRequest and uses it to set starting values for a new builder. The code for builderFrom() follows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/HttpRequest.java

```
public static Builder builderFrom(HttpRequest request) {
        Builder builder = new Builder();
        builder.path(request.getPath());
        builder.body(request.getBody());

        Map<String, String> headers = request.getHeaders();
        for (String headerName : headers.keySet())
                builder.addHeader(headerName,
                                  headers.get(headerName));

        return builder;
}
```

This may seem wasteful, but the JVM is a miracle of modern software engineering. It's able to garbage-collect short-lived objects very efficiently, so this style of programming performs admirably well in most domains.

Views and Strategy

Let's continue our tour of TinyWeb with a look at view handling. In a fully featured framework, we'd include some ways to plug template engines into our view, but for TinyWeb we'll just assume we're generating our response bodies entirely in code using string manipulation.

First we'll need a View interface, which has a single method, render(). render() takes in a model in the form of a Map<String, List<String>>, which represents the

Immutability: Not Just for Functional Programmers

The experienced object-oriented programmer might grumble about extra effort to get immutable objects, especially if we're doing it "just to be functional." However, immutable data doesn't just fall out of functional programming; it's a good practice that can help us write cleaner code.

A large class of software bugs boil down to one section of code modifying data in another section in an unexpected way. This type of bug becomes even more heinous in the multicore world we all live in now. By making our data immutable, we can avoid this class of bugs altogether.

Using immutable data is an oft-repeated bit of advice in the Java world; it's mentioned in *Effective Java [Blo08]*—*Item 15: Minimize Mutability*, among other places, but it is rarely followed. This is largely due to the fact that Java wasn't designed with immutability in mind, so it takes a lot of programmer effort to get it.

Still, some popular, high-quality libraries, such as Joda-Time and Google's collections library, provide excellent support for programming with immutable data. The fact that both of these popular libraries provide replacements for functionality available in Java's standard library speaks to the usefulness of immutable data.

Thankfully, both Scala and Clojure have much more first-class support for immutable data, to the extent that it's often harder to use mutable data than immutable.

model attributes and values. We'll use a List<String> for our values so that a single attribute can have multiple values. It returns a String representing the rendered view.

The View interface is in the following code:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/View.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public interface View {
        public String render(Map<String, List<String>> model);
}
```

Next we need two classes that are designed to work together using the Strategy pattern: StrategyView and RenderingStrategy.

RenderingStrategy is responsible for doing the actual work of rendering a view as implemented by the framework user. It's an instance of a Strategy class from the Strategy pattern, and its code follows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/RenderingStrategy.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public interface RenderingStrategy {

        public String renderView(Map<String, List<String>> model);

}
```

Now let's examine the class that delegates to RenderingStrategy, StrategyView. This class is implemented by the framework and takes care of properly handing exceptions thrown out of the RenderingStrategy. Its code follows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/StrategyView.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public class StrategyView implements View {

        private RenderingStrategy viewRenderer;

        public StrategyView(RenderingStrategy viewRenderer) {
                this.viewRenderer = viewRenderer;
        }

        @Override
        public String render(Map<String, List<String>> model) {
                try {
                        return viewRenderer.renderView(model);
                } catch (Exception e) {
                        throw new RenderingException(e);
                }
        }
}
```

To implement a view, the framework user creates a new subclass of RenderingStrategy with the right view-rendering logic, and the framework injects it into StrategyView.

In this simple example, StrategyView plays a minimal role. It simply swallows exceptions and wraps them in RenderingException so that they can be handled properly at a higher level. A more complete framework might use StrategyView as an integration point for various rendering engines, among other things, but we'll keep it simple here.

Controllers and Template Method

Next up is our Controller. The Controller itself is a simple interface with a single method, handleRequest(), which takes an HttpRequest and returns an HttpResponse. The code for the interface follows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/Controller.java

```
package com.mblinn.oo.tinyweb;

public interface Controller {
        public HttpResponse handleRequest(HttpRequest httpRequest);
}
```

We'll use the Template Method pattern so that users can implement their own controllers. The central class for this implementation is TemplateController, which has an abstract doRequest(), as shown in the following code:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/TemplateController.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public abstract class TemplateController implements Controller {
        private View view;
        public TemplateController(View view) {
                this.view = view;
        }

        public HttpResponse handleRequest(HttpRequest request) {
                Integer responseCode = 200;
                String responseBody = "";

                try {
                        Map<String, List<String>> model = doRequest(request);
                        responseBody = view.render(model);
                } catch (ControllerException e) {
                        responseCode = e.getStatusCode();
                } catch (RenderingException e) {
                        responseCode = 500;
                        responseBody = "Exception while rendering.";
                } catch (Exception e) {
                        responseCode = 500;
                }

                return HttpResponse.Builder.newBuilder().body(responseBody)
                    .responseCode(responseCode).build();
        }
        protected abstract Map<String, List<String>> doRequest(HttpRequest request);
}
```

To implement a controller, a user of the framework extends TemplateController and implements its doRequest() method.

Both the Template Method pattern we used for our controllers and the Strategy pattern we used for our views support similar tasks. They let some general code, perhaps in a library or framework, delegate out to another bit of code intended to perform a specific task. The Template Method pattern does it using inheritance, while the Strategy pattern does it using composition.

In the functional world, we'll rely heavily on composition, which also happens be good practice in the object-oriented world. However, it'll be a composition of functions rather than a composition of objects.

Filter and Functional Interface

Finally, let's examine Filter. The Filter class is a Functional Interface that lets us perform some action on HttpRequest before it's processed. For instance, we may want to log some information about the request or even add a header. It has a single method, doFilter(), takes HttpRequest, and returns a filtered instance of it.

If an individual Filter needs to do something that modifies a request, it simply creates a new one based on the existing request and returns it. This lets us work with an immutable HttpRequest but gives us the illusion that it can be changed.

The code for Filter follows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/Filter.java

```
package com.mblinn.oo.tinyweb;

public interface Filter {
        public HttpRequest doFilter(HttpRequest request);
}
```

Now that we've seen all of the pieces of TinyWeb, let's see how they fit together.

Tying It All Together

To tie it all together, we'll use the main class, TinyWeb. This class takes two constructor arguments. The first is a Map, where the keys are Strings representing request paths and the values are Controller objects. The second argument is a list of Filters to run on all requests before they are passed to the appropriate controller.

The TinyWeb class has a single public method, handleRequest(), which takes HttpRequest. The handleRequest() method then runs the request through the filters, looks up the appropriate controller to handle it, and returns the resulting HttpResponse. The code is below:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/TinyWeb.java

```
package com.mblinn.oo.tinyweb;

import java.util.List;
import java.util.Map;

public class TinyWeb {
        private Map<String, Controller> controllers;
        private List<Filter> filters;

        public TinyWeb(Map<String, Controller> controllers, List<Filter> filters) {
                this.controllers = controllers;
                this.filters = filters;
        }

        public HttpResponse handleRequest(HttpRequest httpRequest) {

                HttpRequest currentRequest = httpRequest;
                for (Filter filter : filters) {
                        currentRequest = filter.doFilter(currentRequest);
                }

                Controller controller = controllers.get(currentRequest.getPath());

                if (null == controller)
                        return null;

                return controller.handleRequest(currentRequest);
        }
}
```

A full-featured Java web framework wouldn't expose a class like this directly as its framework plumbing. Instead it would use some set of configuration files and annotations to wire things together. However, we'll stop adding to TinyWeb here and move on to an example that uses it.

Using TinyWeb

Let's implement an example program that takes an HttpRequest with a comma-separated list of names as its value and returns a body that's full of friendly greetings for those names. We'll also add a filter that logs the path that was requested.

We'll start by looking at GreetingController. When the controller receives an HttpRequest, it picks out the body of the request, splits it on commas, and treats each element in the split body as a name. It then generates a random friendly greeting for each name and puts the names into the model under the key greetings. The code for GreetingController follows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/example/GreetingController.java

```
package com.mblinn.oo.tinyweb.example;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;

import com.mblinn.oo.tinyweb.HttpRequest;
import com.mblinn.oo.tinyweb.TemplateController;
import com.mblinn.oo.tinyweb.View;

public class GreetingController extends TemplateController {
        private Random random;
        public GreetingController(View view) {
                super(view);
                random = new Random();
        }

        @Override
        public Map<String, List<String>> doRequest(HttpRequest httpRequest) {
                Map<String, List<String>> helloModel =
                                new HashMap<String, List<String>>();
                helloModel.put("greetings",
                                generateGreetings(httpRequest.getBody()));
                return helloModel;
        }

        private List<String> generateGreetings(String namesCommaSeperated) {
                String[] names = namesCommaSeperated.split(",");
                List<String> greetings = new ArrayList<String>();
                for (String name : names) {
                        greetings.add(makeGreeting(name));
                }
                return greetings;
        }

        private String makeGreeting(String name) {
                String[] greetings =
                        { "Hello", "Greetings", "Salutations", "Hola" };
                String greetingPrefix = greetings[random.nextInt(4)];
                return String.format("%s, %s", greetingPrefix, name);
        }
}
```

Next up, let's take a look at GreetingRenderingStrategy. This class iterates through the list of friendly greetings generated by the controller and places each into an <h2> tag. Then it prepends the greetings with an <h1> containing "Friendly Greetings:", as the following code shows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/example/GreetingRenderingStrategy.java

```
package com.mblinn.oo.tinyweb.example;

import java.util.List;
import java.util.Map;

import com.mblinn.oo.tinyweb.RenderingStrategy;

public class GreetingRenderingStrategy implements RenderingStrategy {

        @Override
        public String renderView(Map<String, List<String>> model) {
                List<String> greetings = model.get("greetings");
                StringBuffer responseBody = new StringBuffer();
                responseBody.append("<h1>Friendly Greetings:</h1>\n");
                for (String greeting : greetings) {
                        responseBody.append(
                                        String.format("<h2>%s</h2>\n", greeting));

                }
                return responseBody.toString();
        }

}
```

Finally, let's look at an example filter. The LoggingFilter class just logs out the path of the request it's being run on. Its code follows:

JavaExamples/src/main/java/com/mblinn/oo/tinyweb/example/LoggingFilter.java

```
package com.mblinn.oo.tinyweb.example;

import com.mblinn.oo.tinyweb.Filter;
import com.mblinn.oo.tinyweb.HttpRequest;

public class LoggingFilter implements Filter {

        @Override
        public HttpRequest doFilter(HttpRequest request) {
                System.out.println("In Logging Filter - request for path: "
                    + request.getPath());
                return request;
        }

}
```

Wiring up a simple test harness that connects everything together into a TinyWeb, throws an HttpRequest at it, and then prints the response to the console gets us the following output. This indicates that everything is working properly:

```
In Logging Filter - request for path: greeting/
responseCode: 200
responseBody:
<h1>Friendly Greetings:</h1>
<h2>Hola, Mike</h2>
<h2>Greetings, Joe</h2>
<h2>Hola, John</h2>
<h2>Salutations, Steve</h2>
```

Now that we've seen the TinyWeb framework in Java, let's take a look at how we'll use some of the functional replacements for the object-oriented patterns we'll explore in this book. This will give us a TinyWeb that's functionally equivalent but written with fewer lines of code and in a more declarative, easier-to-read style.

2.3 TinyWeb in Scala

Let's take TinyWeb and transform it into Scala. We'll do this a bit at a time so we can show how our Scala code can work with the existing Java code. The overall shape of the framework will be similar to the Java version, but we'll take advantage of some of Scala's functional features to make the code more concise.

Step One: Changing Views

We'll start with our view code. In Java, we used the classic Strategy pattern. In Scala, we'll stick with the Strategy pattern, but we'll use *higher-order functions* for our strategy implementations. We'll also see some of the benefits of expressions over statements for control flow.

The biggest change we'll make is to the view-rendering code. Instead of using Functional Interface in the form of RenderingStrategy, we'll use a higher-order function. We go over this replacement in great detail in Pattern 1, *Replacing Functional Interface*, on page ?.

Here's our modified view code in its full functional glory:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepone/View.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepone
import com.mblinn.oo.tinyweb.RenderingException

trait View {
  def render(model: Map[String, List[String]]): String
}
```

```
class FunctionView(viewRenderer: (Map[String, List[String]]) => String)
        extends View {
  def render(model: Map[String, List[String]]) =
    try
        viewRenderer(model)
    catch {
        case e: Exception => throw new RenderingException(e)
    }
}
```

We start off with our View trait. It defines a single method, render(), which takes a map representing the data in our model and returns a rendered String.

```
trait View {
  def render(model: Map[String, String]): String
}
```

Next up, let's take a look at the body of FunctionView. The code below declares a class that has a constructor with a single argument, viewRenderer, which sets an immutable field of the same name.

```
class FunctionView(viewRenderer: (Map[String, String]) => String) extends View {
  «classBody»
}
```

The viewRenderer function parameter has a rather strange-looking type annotation, (Map[String, String]) => String. This is a function type. It says that viewRenderer is a function that takes a Map[String, String] and returns a String, just like the renderView() on our Java RenderingStrategy.

Next, let's take a look at the render() method itself. As we can see from the code below, it takes in a model and runs it through the viewRender() function.

```
def render(model: Map[String, String]) =
  try
    viewRenderer(model)
  catch {
    case e: Exception => throw new RenderingException(e)
  }
```

Notice how there's no return keyword anywhere in this code snippet? This illustrates another important aspect of functional programming. In the functional world, we program primarily with expressions. The value of a function is just the value of the last expression in it.

In this example, that expression happens to be a try block. If no exception is thrown, the try block takes on the value of its main branch; otherwise it takes on the value of the appropriate case clause in the catch branch.

If we wanted to supply a default value rather than wrap the exception up into a RenderException, we can do so just by having the appropriate case branch take on our default, as illustrated in the following code:

```
try
  viewRenderer(model)
catch {
  case e: Exception => ""
}
```

Now when an exception is caught, the try block takes on the value of the empty string.

Step Two: A Controller First Cut

Now let's take a look at transforming our controller code into Scala. In Java we used the Controller interface and the TemplateController class. Individual controllers were implemented by subclassing TemplateController.

In Scala, we rely on function composition just like we did with our views by passing in a doRequest() function when we create a Controller:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/steptwo/Controller.scala

```
package com.mblinn.mbfpp.oo.tinyweb.steptwo

import com.mblinn.oo.tinyweb.HttpRequest
import com.mblinn.oo.tinyweb.HttpResponse
import com.mblinn.oo.tinyweb.ControllerException
import com.mblinn.oo.tinyweb.RenderingException

trait Controller {
  def handleRequest(httpRequest: HttpRequest): HttpResponse
}

class FunctionController(view: View, doRequest: (HttpRequest) =>
  Map[String, List[String]] ) extends Controller {

  def handleRequest(request: HttpRequest): HttpResponse = {
    var responseCode = 200;
    var responseBody = "";

    try {
      val model = doRequest(request)
      responseBody = view.render(model)
    } catch {
      case e: ControllerException =>
        responseCode = e.getStatusCode()
      case e: RenderingException =>
        responseCode = 500
        responseBody = "Exception while rendering."
```

```
      case e: Exception =>
        responseCode = 500
    }

    HttpResponse.Builder.newBuilder()
                    .body(responseBody).responseCode(responseCode).build()
  }
}
```

This code should look fairly similar to our view code. This is a fairly literal translation of Java into Scala, but it's not terribly functional because we're using the try-catch as a statement to set the values of responseCode and responseBody.

We're also reusing our Java HttpRequest and HttpResponse. Scala provides a more concise way to create these data-carrying classes, called *case classes*. Switching over to use the try-catch as a statement, as well as using case classes, can help cut down on our code significantly.

We'll make both of these changes in our next transformation.

Immutable HttpRequest and HttpResponse

Let's start by switching over to case classes instead of using the Builder pattern. It's as simple as the code below:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepthree/HttpData.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepthree

case class HttpRequest(headers: Map[String, String], body: String, path: String)
case class HttpResponse(body: String, responseCode: Integer)
```

We can create new HttpRequest and HttpResponse objects easily, as the following REPL output shows:

```
scala> val request = HttpRequest(Map("X-Test" -> "Value"), "requestBody", "/test")
request: com.mblinn.mbfpp.oo.tinyweb.stepfour.HttpRequest =
        HttpRequest(Map(X-Test -> Value),requestBody,/test)

scala> val response = HttpResponse("requestBody", 200)
response: com.mblinn.mbfpp.oo.tinyweb.stepfour.HttpResponse =
        HttpResponse(requestBody,200)
```

At first glance, this might seem similar to using a Java class with constructor arguments, except that we don't need to use the new keyword. However, in Pattern 4, *Replacing Builder for Immutable Object*, on page ?, we dig deeper and see how Scala's ability to provide default arguments in a constructor, the natural immutability of case classes, and the ability to easily create a new instance of a case class from an existing instance lets them satisfy the intent of the Builder pattern.

Let's take a look at our second change. Since a try-catch block in Scala has a value, we can use it as an expression rather than as a statement. This might seem a bit odd at first, but the upshot is that we can use the fact that Scala's try-catch is an expression to simply have the try-catch block take on the value of the HttpResponse we're returning. The code to do so is below:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepthree/Controller.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepthree
import com.mblinn.oo.tinyweb.ControllerException
import com.mblinn.oo.tinyweb.RenderingException

trait Controller {
  def handleRequest(httpRequest: HttpRequest): HttpResponse
}
class FunctionController(view: View, doRequest: (HttpRequest) =>
  Map[String, List[String]] ) extends Controller {
  def handleRequest(request: HttpRequest): HttpResponse =
    try {
      val model = doRequest(request)
      val responseBody = view.render(model)
      HttpResponse(responseBody, 200)
    } catch {
      case e: ControllerException =>
        HttpResponse("", e.getStatusCode)
      case e: RenderingException =>
        HttpResponse("Exception while rendering.", 500)
      case e: Exception =>
        HttpResponse("", 500)
    }
}
```

This style of programming has a couple of benefits. First, we've eliminated a couple of extraneous variables, responseCode and responseBody. Second, we've reduced the number of lines of code a programmer needs to scan to understand which HttpRequest we're returning from the entire method to a single line.

Rather than tracing the values of responseCode and responseBody from the top of the method through the try block and finally into the HttpResponse, we only need to look at the appropriate piece of the try block to understand the final value of the HttpResponse. These changes combine to give us code that's more readable and concise.

Tying It Together

Now let's add in the class that ties it all together, TinyWeb. Like its Java counterpart, TinyWeb is instantiated with a map of Controllers and a map of filters. Unlike Java, we don't define a class for filter; we simply use a list of higher-order functions!

Also like the Java version, the Scala TinyWeb has a single method, handleRequest(), which takes in an HttpRequest. Instead of returning an HttpResponse directly, we return an Option[HttpResponse], which gives us a clean way of handling the case when we can't find a controller for a particular request. The code for the Scala TinyWeb is below:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/stepfour/Tinyweb.scala

```
package com.mblinn.mbfpp.oo.tinyweb.stepfour
class TinyWeb(controllers: Map[String, Controller],
    filters: List[(HttpRequest) => HttpRequest]) {

  def handleRequest(httpRequest: HttpRequest): Option[HttpResponse] = {
    val composedFilter = filters.reverse.reduceLeft(
        (composed, next) => composed compose next)
    val filteredRequest = composedFilter(httpRequest)
    val controllerOption = controllers.get(filteredRequest.path)
    controllerOption map { controller => controller.handleRequest(filteredRequest) }
  }
}
```

Let's take a look at it in greater detail starting with the class definition.

```
class TinyWeb(controllers: Map[String, Controller],
    filters: List[(HttpRequest) => HttpRequest]) {
        «classBody»
}
```

Here we're defining a class that takes two constructor arguments, a map of controllers and a list of filters. Note the type of the filters argument, List[(HttpRequest) => HttpRequest]. This says that filters is a list of functions from HttpRequest to HttpRequest.

Next up, let's look at the signature of the handleRequest() method:

```
def handleRequest(httpRequest: HttpRequest): Option[HttpResponse] = {
        «functionBody»
}
```

As advertised, we're returning an Option[HttpResponse] instead of an HttpResponse. The Option type is a container type with two subtypes, Some and None. If we've got a value to store in it, we can store it in an instance of Some; otherwise we use None to indicate that we've got no real value. We'll cover Option in greater detail in Pattern 8, *Replacing Null Object*, on page ?.

Now that we've seen the TinyWeb framework, let's take a look at it in action. We'll use the same example from the Java section, returning a list of friendly greetings. However, since it's Scala, we can poke at our example in the REPL as we go. Let's get started with our view code.

Using Scala TinyWeb

Let's take a look at using our Scala TinyWeb framework.

We'll start by creating a FunctionView and the rendering function we'll compose into it. The following code creates this function, which we'll name greetingViewRenderer(), and the FunctionView that goes along with it:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala

```
def greetingViewRenderer(model: Map[String, List[String]]) =
  "<h1>Friendly Greetings:%s".format(
      model
      getOrElse("greetings", List[String]())
      map(renderGreeting)
      mkString ", ")

private def renderGreeting(greeting: String) =
  "<h2>%s</h2>".format(greeting)

def greetingView = new FunctionView(greetingViewRenderer)
```

We're using a couple of new bits of Scala here. First, we introduce the map() method, which lets us map a function over all the elements in a sequence and returns a new sequence. Second, we're using a bit of syntactic sugar that Scala provides that allows us to treat any method with a single argument as an infix operator. The object on the left side of the operator is treated as the receiver of the method call, and the object on the right is the argument.

This bit of syntax means that we can omit the familiar dot syntax when working in Scala. For instance, the two usages of map() below are equivalent:

```
scala> val greetings = List("Hi!", "Hola", "Aloha")
greetings: List[java.lang.String]

scala> greetings.map(renderGreeting)
res0: List[String] = List(<h2>Hi!</h2>, <h2>Hola</h2>, <h2>Aloha</h2>)

scala> greetings map renderGreeting
res1: List[String] = List(<h2>Hi!</h2>, <h2>Hola</h2>, <h2>Aloha</h2>)
```

Now let's take a look at our controller code. Here we create the handleGreetingRequest() function to pass into our Controller. As a helper, we use makeGreeting(), which takes in a name and generates a random friendly greeting.

Inside of handleGreetingRequest() we create a list of names by splitting the request body, which returns an array like in Java, converting that array into a Scala list and mapping the makeGreeting() method over it. We then use that list as the value for the "greetings" key in our model map:

Scala Functions and Methods

Since Scala is a hybrid language, it's got both functions and methods. Methods are defined using the def keyword, as we do in the following code snippet:

```
scala> def addOneMethod(num: Int) = num + 1
addOneMethod: (num: Int)Int
```

We can create a function and name it by using Scala's anonymous function syntax, assigning the resulting function to a val, like we do in this code snippet:

```
scala> val addOneFunction = (num: Int) => num + 1
addOneFunction: Int => Int = <function1>
```

We can almost always use methods as higher-order functions. For instance, here we pass both the method and the function version of addOne() into map().

```
scala> val someInts = List(1, 2, 3)
someInts: List[Int] = List(1, 2, 3)

scala> someInts map addOneMethod
res1: List[Int] = List(2, 3, 4)

scala> someInts map addOneFunction
res2: List[Int] = List(2, 3, 4)
```

Since method definitions have a cleaner syntax, we use them when we need to define a function, rather than using the function syntax. When we need to manually convert a method into a function, we can do so with the underscore operator, as we do in the following REPL session:

```
scala> addOneMethod _
res3: Int => Int = <function1>
```

The need to do this is very rare, though; for the most part Scala is smart enough to do the conversion automatically.

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala

```
def handleGreetingRequest(request: HttpRequest) =
  Map("greetings" -> request.body.split(",").toList.map(makeGreeting))

private def random = new Random()
private def greetings = Vector("Hello", "Greetings", "Salutations", "Hola")
private def makeGreeting(name: String) =
  "%s, %s".format(greetings(random.nextInt(greetings.size)), name)

def greetingController = new FunctionController(greetingView, handleGreetingRequest)
```

Finally, let's take a look at our logging filter. This function simply writes the path that it finds in the passed-in HttpRequest to the console and then returns the path unmodified:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala
```
private def loggingFilter(request: HttpRequest) = {
  println("In Logging Filter - request for path: %s".format(request.path))
  request
}
```

To finish up the example, we need to create an instance of TinyWeb with the controller, the view, and the filter we defined earlier, and we need to create a test HttpResponse:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/oo/tinyweb/example/Example.scala
```
def tinyweb = new TinyWeb(
    Map("/greeting" -> greetingController),
    List(loggingFilter))
def testHttpRequest = HttpRequest(
    body="Mike,Joe,John,Steve",
    path="/greeting")
```

We can now run the test request through TinyWeb's handleRequest() method in the REPL and view the corresponding HttpResponse:

```
scala> tinyweb.handleRequest(testHttpRequest)
In Logging Filter - request for path: /greeting
res0: Option[com.mblinn.mbfpp.oo.tinyweb.stepfour.HttpResponse] =
Some(HttpResponse(<h1>Friendly Greetings:<h2>Mike</h2>, <h2>Nam</h2>, <h2>John</h2>,
200))
```

That wraps up our Scala version of TinyWeb. We've made a few changes to the style that we used in our Java version. First, we replaced most of our iterative code with code that's more declarative. Second, we've replaced our bulky builders with Scala's case classes, which give us a built-in way to handle immutable data. Finally, we've replaced our use of Functional Interface with plain old functions.

Taken together, these small changes save us quite a bit of code and give us a solution that's shorter and easier to read. Next up, we'll take a look at TinyWeb in Clojure.