http://www.yegor256.com/2015/03/22/takes-java-web-framework.html

Java Web App Architecture In Takes Framework

22 March 2015 modified on 5 April 2015 Yegor Bugayenko

I used to utilize Servlets, JSP, JAX-RS, Spring Framework, Play Framework, JSF with Facelets, and a bit of Spark Framework. All of these solutions, in my humble opinion, are very far from being object-oriented and elegant. They all are full of static methods, untestable data structures, and dirty hacks. So about a month ago, I decided to create my own Java web framework. I put a few basic principles into its foundation: 1) No NULLs, 2) no public static methods, 3) no mutable classes, and 4) no class casting, reflection, and `instanceof` operators. These four basic principles should guarantee clean code and transparent architecture. That's how the Takes framework was born. Let's see what was created and how it works.

© Making of The Godfather (1972) by Francis Ford Coppola

Java Web Architecture in a Nutshell

This is how I understand a web application architecture and its components, in simple terms.

First, to create a web server, we should create a new network socket, that accepts connections on a certain TCP port. Usually it is 80, but I'm going to use 8080 for testing purposes. This is done in Java with the `ServerSocket` class:

```
import java.net.ServerSocket;
public class Foo {
  public static void main(final String... args) throws Exception {
    final ServerSocket server = new ServerSocket(8080);
    while (true);
  }
}
```

That's enough to start a web server. Now, the socket is ready and listening on port 8080. When someone opens `http://localhost:8080` in their browser, the connection will be established and the browser will spin its waiting wheel forever. Compile this snippet and try. We just built a simple web server without the use of any frameworks. We're not doing anything with incoming connections yet, but we're not rejecting them either. All of them are being lined up inside that `server` object. It's being done in a background thread; that's why we need to put that `while(true)` in afterward. Without this endless pause, the app will finish its execution immediately and the server socket will shut down.

The next step is to accept the incoming connections. In Java, that's done through a blocking call to the `accept()` method:

```
final Socket socket = server.accept();
```

The method is blocking its thread and waiting until a new connection arrives. As soon as that happens, it returns an instance of `Socket`. In order to accept the next connection, we should call `accept()` again. So basically, our web server should work like this:

```
public class Foo {
  public static void main(final String... args) throws Exception {
    final ServerSocket server = new ServerSocket(8080);
    while (true) {
      final Socket socket = server.accept();
      // 1. Read HTTP request from the socket
      // 2. Prepare an HTTP response
      // 3. Send HTTP response to the socket
      // 4. Close the socket
    }
  }
}
```

It's an endless cycle that accepts a new connection, understands it, creates a response, returns the response, and accepts a new connection again. HTTP protocol is stateless, which means the server should not remember what happened in any previous connection. All it cares about is the incoming HTTP request in this particular connection.

The HTTP request is coming from the input stream of the socket and looks like a multi-line block of text. This is what you would see if you read an input stream of the socket:

```
final BufferedReader reader = new BufferedReader(
  new InputStreamReader(socket.getInputStream())
);
while (true) {
  final String line = reader.readLine();
  if (line.isEmpty()) {
    break;
  }
  System.out.println(line);
}
```

You will see something like this:

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWeb
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,ru;q=0.6,uk;q=0.4
```

The client (the Google Chrome browser, for example) passes this text into the connection established. It connects to port 8080 at `localhost`, and as soon as the connection is ready, it immediately sends this text into it, then waits for a response.

Our job is to create an HTTP response using the information we get in the request. If our server is very primitive, we can basically ignore all the information in the request and just return "Hello, world!" to all requests (I'm using `IOUtils` for simplicity):

```
import java.net.Socket;
import java.net.ServerSocket;
import org.apache.commons.io.IOUtils;
public class Foo {
  public static void main(final String... args) throws Exception {
    final ServerSocket server = new ServerSocket(8080);
    while (true) {
      try (final Socket socket = server.accept()) {
        IOUtils.copy(
          IOUtils.toInputStream("HTTP/1.1 200 OK\r\n\r\nHello, worl
          socket.getOutputStream()
        );
      }
    }
  }
}
```

That's it. The server is ready. Try to compile and run it. Point your browser to http://localhost:8080, and you will see `Hello, world!` :

```
$ javac -cp commons-io.jar Foo.java
$ java -cp commons-io.jar:. Foo &
$ curl http://localhost:8080 -v
* Rebuilt URL to: http://localhost:8080/
* Connected to localhost (::1) port 8080 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
* no chunk, no close, no size. Assume close to signal end
<
* Closing connection 0
Hello, world!
```

That's all you need to build a web server. Now let's discuss how to make it object-oriented and composable. Let's try to see how the Takes framework was built.

Routing/Dispatching

Routing/dispatching is combined with response printing in Takes. All you need to do to create a working web application is to create a single class that implements `Take` interface:

```
import org.takes.Request;
import org.takes.Take;
public final class TkFoo implements Take {
  @Override
  public Response route(final Request request) {
    return new RsText("Hello, world!");
  }
```

```
}
```

And now it's time to start a server:

```
import org.takes.http.Exit;
import org.takes.http.FtBasic;
public class Foo {
  public static void main(final String... args) throws Exception {
    new FtBasic(new TkFoo(), 8080).start(Exit.NEVER);
  }
}
```

This `FtBasic` class does the exact same socket manipulations explained above. It starts a server socket on port 8080 and dispatches all incoming connections through an instance of `TkFoo` that we are giving to its constructor. It does this dispatching in an endless cycle, checking every second whether it's time to stop with an instance of `Exit`. Obviously, `Exit.NEVER` always responds with, "Don't stop, please".

HTTP Request

Now let's see what's inside the HTTP request arriving at `TsFoo` and what we can get out of it. This is how the `Request` interface is defined in Takes:

```
public interface Request {
  Iterable<String> head() throws IOException;
  InputStream body() throws IOException;
}
```

The request is divided into two parts: the head and the body. The head contains all lines that go before the empty line that starts a body, according to HTTP specification in RFC 2616. There are many useful decorators for `Request` in the framework. For example, `RqMethod` will help you get the method name from the first line of the header:

```
final String method = new RqMethod(request).method();
```

`RqHref` will help extract the query part and parse it. For example, this is the request:

```
GET /user?id=123 HTTP/1.1
Host: www.example.com
```

This code will extract that `123`:

```
final int id = Integer.parseInt(
  new RqHref(request).href().param("id").get(0)
);
```

`RqPrint` can get the entire request or its body printed as a `String`:

```
final String body = new RqPrint(request).printBody();
```

The idea here is to keep the `Request` interface simple and provide this request parsing functionality to its decorators. This approach helps the framework keep classes small and cohesive. Each decorator is very small and solid, doing exactly one thing. All of these decorators are in the `org.takes.rq` package. As you already probably understand, the `Rq` prefix stands for `Request`.

First Real Web App

Let's create our first real web application, which will do something useful. I would recommend starting with an `Entry` class, which is required by Java to start an app from the command line:

```
import org.takes.http.Exit;
import org.takes.http.FtCLI;
public final class Entry {
  public static void main(final String... args) throws Exception {
    new FtCLI(new TkApp(), args).start(Exit.NEVER);
  }
}
```

This class contains just a single `main()` static method that will be called by JVM when the app starts from the command line. As you see, it instantiates `FtCLI`, giving it an instance of class `TkApp` and command line arguments. We'll create the `TkApp` class in a second. `FtCLI` (translates to "front-end with command line interface") makes an instance of the same `FtBasic`, wrapping it into a few useful decorators and configuring it according to command line arguments. For example, `--port=8080` will be converted into a `8080` port number and passed as a second argument of the `FtBasic` constructor.

The web application itself is called `TkApp` and extends `TsWrap`:

```
import org.takes.Take;
import org.takes.facets.fork.FkRegex;
import org.takes.facets.fork.TkFork;
import org.takes.tk.TkWrap;
import org.takes.tk.TkClasspath;
final class TkApp extends TkWrap {
  TkApp() {
    super(TkApp.make());
  }
  private static Take make() {
    return new TkFork(
      new FkRegex("/robots.txt", ""),
      new FkRegex("/css/.*", new TkClasspath()),
      new FkRegex("/", new TkIndex())
    );
  }
}
```

We'll discuss this `TkFork` class in a minute.

If you're using Maven, this is the `pom.xml` you should start with:

```
<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven
  <modelVersion>4.0.0</modelVersion>
  <groupId>foo</groupId>
  <artifactId>foo</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.takes</groupId>
      <artifactId>takes</artifactId>
      <version>0.9</version> <!-- check the latest in Maven Central
    </dependency>
  </dependencies>
  <build>
    <finalName>foo</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
            <configuration>
              <outputDirectory>${project.build.directory}/deps</outp
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Running `mvn clean package` should build a `foo.jar` file in `target` directory and a collection of all JAR dependencies in `target/deps`. Now you can run the app from the command line:

```
$ mvn clean package
$ java -Dfile.encoding=UTF-8 -cp ./target/foo.jar:./target/deps/* fo
```

The application is ready, and you can deploy it to, say, Heroku. Just create a `Procfile` file in the root of the repository and push the repo to Heroku. This is what `Procfile` should look like:

```
web: java -Dfile.encoding=UTF-8 -cp target/foo.jar:target/deps/* foo
```

TkFork

This `TkFork` class seems to be one of the core elements of the framework. It helps route an incoming HTTP request to the right *take*. Its logic is very simple, and there are just a few lines of code inside it. It encapsulates a collection of "forks", which are instances of the `Fork` interface:

```
public interface Fork {
  Iterator<Response> route(Request req) throws IOException;
}
```

Its only `route()` method either returns an empty iterator or an iterator with a single `Response`. `TkFork` goes through all forks, calling their `route()` methods until one of them returns a response. Once that happens, `TkFork` returns this response to the caller, which is `FtBasic`.

Let's create a simple fork ourselves now. For example, we want to show the status of the application when the `/status` URL is requested. Here is the code:

```
final class TkApp extends TkWrap {
  private static Take make() {
    return new TkFork(
      new Fork() {
        @Override
        public Iterator<Response> route(Request req) {
          final Collection<Response> responses = new ArrayList<>(1);
          if (new RqHref(req).href().path().equals("/status")) {
            responses.add(new TkStatus());
          }
          return responses.iterator();
        }
      }
    );
  }
}
```

I believe the logic here is clear. We either return an empty iterator or an iterator with an instance of `TkStatus` inside. If an empty iterator is returned, `TkFork` will try to find another fork in the collection that actually gets an instance of `Response`. By the way, if nothing is found and all forks return empty iterators, `TkFork` will throw a "Page not found" exception.

This exact logic is implemented by an out-of-the-box fork called `FkRegex`, which attempts to match a request URI path with the regular expression provided:

```
final class TkApp extends TkWrap {
  private static Take make() {
    return new TkFork(
      new FkRegex("/status", new TkStatus())
    );
  }
}
```

We can compose a multi-level structure of `TkFork` classes; for example:

```
final class TkApp extends TsWrap {
  private static Take make() {
    return new TkFork(
      new FkRegex(
        "/status",
        new TkFork(
          new FkParams("f", "json", new TkStatusJSON()),
          new FkParams("f", "xml", new TkStatusXML())
        )
      )
    );
  }
}
```

Again, I believe it's obvious. The instance of `FkRegex` will ask an encapsulated instance of `TkFork` to return a response, and it will try to fetch it from one that `FkParams` encapsulated. If the HTTP query is `/status?f=xml`, an instance of `TkStatusXML` will be returned.

HTTP Response

Now let's discuss the structure of the HTTP response and its object-oriented abstraction, `Response`. This is how the interface looks:

```
public interface Response {
  Iterable<String> head() throws IOException;
  InputStream body() throws IOException;
}
```

Looks very similar to the `Request`, doesn't it? Well, it's identical, mostly because the structure of the HTTP request and response is almost identical. The only difference is the first line.

There is a collection of useful decorators that help in response building. They are composable, which makes them very convenient. For example, if you want to build a response that contains an HTML page, you compose them like this:

```
final class TkIndex implements Take {
  @Override
  public Response act() {
    return new RsWithStatus(
      new RsWithType(
        new RsWithBody("<html>Hello, world!</html>"),
        "text/html"
      ),
      200
    );
  }
}
```

In this example, the decorator `RsWithBody` creates a response with a body but with no headers at all. Then, `RsWithType` adds the header `Content-Type: text/html` to it. Then, `RsWithStatus` makes sure the first line of the response contains `HTTP/1.1 200 OK`.

You can create your own decorators that can reuse existing ones. Take a look at how it's done in [`RsPage`] from rultor.com.

How About Templates?

Returning simple "Hello, world" pages is not a big problem, as we can see. But what about more complex output like HTML pages, XML documents, JSON data sets, etc? There are a few convenient `Response` decorators that enable all of that. Let's start with Velocity, a simple templating engine. Well, it's not that simple. It's rather powerful, but I would suggest to use it in simple situations only. Here is how it works:

```
final class TkIndex implements Take {
  @Override
  public Response act() {
    return new RsVelocity("Hello, ${name}")
```

```
      .with("name", "Jeffrey");
  }
}
```

The `RsVelocity` constructor accepts a single argument that has to be a Velocity template. Then, you call the `with()` method, injecting data into the Velocity context. When it's time to render the HTTP response, `RsVelocity` will "evaluate" the template against the context configured. Again, I would recommend you use this templating approach only for simple outputs.

For more complex HTML documents, I would recommend you use XML/XSLT in combination with Xembly. I explained this idea in a few previous posts: XML+XSLT in a Browser and RESTful API and a Web Site in the Same URL. It is simple and powerful — Java generates XML output and the XSLT processor transforms it into HTML documents. This is how we separate representation from data. The XSL stylesheet is a "view" and `TkIndex` is a "controller", in terms of MVC.

I'll write a separate article about templating with Xembly and XSL very soon.

In the meantime, we'll create decorators for JSF/Facelets and JSP rendering in Takes. If you're interested in helping, please fork the framework and submit your pull requests.

What About Persistence?

Now, a question that comes up is what to do with persistent entities, like databases, in-memory structures, network connections, etc. My suggestion is to initialize them inside the `Entry` class and pass them as arguments into the `TkApp` constructor. Then, the `TkApp` will pass them into the constructors of custom *takes*.

For example, we have a PostgreSQL database that contains some table data that we need to render. Here is how I would initialize a connection to it in the `Entry` class (I'm using a BoneCP connection pool):

```
public final class Entry {
  public static void main(final String... args) throws Exception {
    new FtCLI(new TkApp(Entry.postgres()), args).start(Exit.NEVER);
  }
  private static Source postgres() {
    final BoneCPDataSource src = new BoneCPDataSource();
    src.setDriverClass("org.postgresql.Driver");
    src.setJdbcUrl("jdbc:postgresql://localhost/db");
    src.setUser("root");
    src.setPassword("super-secret-password");
    return src;
  }
}
```

Now, the constructor of `TkApp` must accept a single argument of type `java.sql.Source`:

```
final class TkApp extends TkWrap {
  TkApp(final Source source) {
    super(TkApp.make(source));
  }
  private static Take make(final Source source) {
    return new TkFork(
      new FkRegex("/", new TkIndex(source))
    );
  }
}
```

Class `TkIndex` also accepts a single argument of class `Source`. I believe you know what to do with it inside `TkIndex` in order to fetch the SQL table data and convert it into HTML. The point here is that the dependency must be injected into the application (instance of class `TkApp`) at the moment of its instantiation. This is a pure and clean dependency injection mechanism, which is absolutely container-free. Read more about it in "Dependency Injection Containers Are Code Polluters".

Unit Testing

Since every class is immutable and all dependencies are injected only through constructors, unit testing is extremely easy. Let's say we want to test `TkStatus` , which is supposed to return an HTML response (I'm using JUnit 4 and Hamcrest):

```
import org.junit.Test;
import org.hamcrest.MatcherAssert;
import org.hamcrest.Matchers;
public final class TkIndexTest {
  @Test
  public void returnsHtmlPage() throws Exception {
    MatcherAssert.assertThat(
      new RsPrint(
        new TkStatus().act(new RqFake())
      ).printBody(),
      Matchers.equalsTo("<html>Hello, world!</html>")
    );
  }
}
```

Also, we can start the entire application or any individual *take* in a test HTTP server and test its behavior via a real TCP socket; for example (I'm using jcabi-http to make an HTTP request and check the output):

```
public final class TkIndexTest {
  @Test
  public void returnsHtmlPage() throws Exception {
    new FtRemote(new TkIndex()).exec(
      new FtRemote.Script() {
        @Override
        public void exec(final URI home) throws IOException {
          new JdkRequest(home)
            .fetch()
```

```
          .as(RestResponse.class)
          .assertStatus(HttpURLConnection.HTTP_OK)
          .assertBody(Matchers.containsString("Hello, world!"));
      }
    }
  );
 }
}
```

`FtRemote` starts a test web server at a random TCP port and calls the `exec()` method at the provided instance of `FtRemote.Script`. The first argument of this method is a URI of the just-started web server homepage.

The architecture of Takes framework is very modular and composable. Any individual *take* can be tested as a standalone component, absolutely independent from the framework and other *takes*.

Why the Name?

That's the question I've been hearing rather often. The idea is simple, and it originates from the movie business. When a movie is made, the crew shoots many *takes* in order to capture the reality and put it on film. Each capture is called a *take*.

In other words, a *take* is like a snapshot of the reality.

The same applies to this framework. Each instance of `Take` represents a reality at one particular moment in time. This reality is then sent to the user in the form of a `Response`.