FRAMEWORKS

Javalin: A Simple, Modern Web Server Framework

Building web apps with a fast, lightweight, unopinionated framework that creates tiny executables

by David Åse

Javalin is a very lightweight web framework for Java 8 (and later) and Kotlin. It supports modern features such as HTTP/2, WebSocket, and asynchronous requests. Javalin is servlet-based, and its main goals are simplicity, a great developer experience, and first-class interoperability between Java and Kotlin.

In this article, I explain what Javalin is and how easily it enables you to write web applications quickly. You'll need some experience with the basics of web applications to follow along.

Many developers would say Javalin is a library rather than a framework. This is because in Javalin, unlike in most frameworks, you never extend anything; it sets no requirements for your application structure; and there are no annotations, no reflection, and no other magic—just code. The "Hello World" example is just four lines and an `import` statement:

```java
import io.javalin.Javalin;

public static void main(String[] args) {
    Javalin app = Javalin.create().start(7000);
    app.get("/", ctx -> ctx.result("Hello World"));
}
```

This snippet creates a new Javalin instance and starts it on port 7000. It then attaches a `Handler` that is triggered by GET requests to the root path. You can build and package this application as a JAR file. If you use Maven, just add this to your build:

```xml
<dependency>
    <groupId>io.javalin</groupId>
    <artifactId>javalin</artifactId>
    <version>2.5.0</version>
</dependency>
```

Run the output JAR file like any other Java program (
`java -jar myjar.jar`).

Getting Started

All Javalin programs require the creation of a Javalin instance (
`Javalin.create()`), which creates a web server to which you can attach `Handler` objects. The `Handler` interface has a single method, `handle`, which is void and takes a `Context` as its only parameter. This `Context` contains everything you need for operating on the HTTP request and response.

```
@FunctionalInterface
public interface Handler {
    void handle(Context ctx) throws Exception;
}
```

A `Handler` is attached to the `Javalin` instance with a verb and a path:

```
app.get("/hello-get", ctx -> ctx.result("Hello GET"
```

Responses are set on the `Context` instance (`ctx`) through the `ctx.result()` method. As mentioned previously, the `Context` contains all the methods required to deal with both requests and responses.

You could also write the previous code snippet by creating a class that implements `Handler`:

```
class MyGetHandler implements Handler {
    @Override
    public void handle(Context ctx) {
        ctx.result("Hello GET")
    }
}
```

Then, you need to add an instance of `MyGetHandler` to the `Javalin` instance:

```
app.get("/hello-get", new MyGetHandler());
```

Although it’s possible to write Javalin applications this way, it’s recommended that you use lambda syntax instead. If you need to split up your code, the best approach is to create method references:

```
app.get("/hello-get", helloController::myGetHandler
```

This approach makes it easier to group common functionality and puts fewer restrictions on how you build your application. I’ll present more information on handlers later in this article. Now let’s look at how to handle common operations with Javalin.

JSON responses. A common use case for Javalin is to serve a JSON object. This can be done easily by calling `ctx.json(myObject)`:

```
app.get("/json", ctx -> ctx.json(myObject));
```

This code transforms `myObject` to JSON by using Javalin’s JSON plugin and sets the content type of the response to `application/json`. The JSON plugin is fully configurable, so any JSON library can be used with Javalin—be it Jackson, Gson, or another choice.

Note that Javalin defines two interfaces for mapping to and from JSON and includes a Jackson implementation for both of these interfaces. But you’re free to provide your own implementation to replace the ones that ship with Javalin.

Handling input. All client input is available through the `Context`. You get parameters from the path, the query string, or the request body. The request body can contain either the form parameters or a string (usually JSON). Javalin handles all cases in a consistent way:

```
app.get("/:path-param", ctx -> {
    String qp = ctx.queryParam("query-param");
    String pp = ctx.pathParam("path-param");
    String fp = ctx.formParam("form-param");
    String body = ctx.body();
    MyObject mo = ctx.bodyAsClass(MyObject.class);
});
```

Getting input as a string is great for quick prototyping and debugging, but usually you'll want a specific type of object. For that, you can use the `Validator` class:

```
int index = ctx.validatedQueryParam("index").asInt(
```

In this code, the call to `getOrThrow()` tells Javalin to convert the string to the specified type or throw an exception. These exceptions are automatically mapped to standard HTTP responses, and they provide helpful debug messages to the client. For example, if the `index` query-param is `abc`, the client will be sent the following error:

```
Query parameter 'number' with value 'abc' is not a
```

The `Validator` class also supports an `asClass` method that you can use to validate any type. This enables you to do powerful things, such as validating two `Instant` types relative to each other:

```
Instant fromDate = ctx.validatedQueryParam("from")
    .asClass(Instant.class)
    .getOrThrow();
Instant toDate = ctx.validatedQueryParam("to")
    .asClass(Instant.class)
    .check(to -> to.isAfter(fromDate), "'to' has to
    .getOrThrow();
```

If you were to use `asClass(UnfamiliarType.class)`, Javalin will ask you to register a converter for that particular class.

Filters and mappers. Sometimes you need to apply the same logic for multiple endpoints, or you need to handle errors in a consistent way. These kinds of problems are solved in Javalin by filters and mappers. Just like HTTP endpoints, the filters in Javalin use the `Handler` interface. Filters can be attached to the Javalin instance with or without specifying a path. For example:

```
app.before("/some-path", ctx -> {
    // runs before requests to /some-path
});

app.after(ctx -> {
    // runs after all requests
});
```

The `before` filters run before endpoint handlers. If you want to prevent an endpoint handler from doing something in certain cases, you can throw an exception in a `before` filter. The `after` filter runs after the endpoint handlers (even after exceptions have been handled).

Exception mappers. It's common to throw exceptions when writing controllers for web applications. If a resource is not found, or if a user isn't authorized to view a resource, you throw an exception and handle it elsewhere. Javalin has an exception mapper that lets you map any exception, and it has a set of premapped exceptions for your convenience, such as `BadRequestResponse`, `NotFoundReponse`, and

`UnauthorizedResponse`. Like HTTP endpoint handlers and filters, the exception mapper has access to the `Context`:

```
app.exception(NullPointerException.class, (exceptio
    // handle null pointers here
});
```

WebSocket. Javalin offers a high-level lambda-based WebSocket API:

```
app.ws("/websocket/:path", ws -> {
    ws.onConnect(session -> System.out.println("Con
});
```

The `ws` object is a `WsHandler` and supports the most common WebSocket events:

```
onConnect(WsSession session)
onMessage(WsSession session, String msg)
onMessage(WsSession session, Byte[] msg, int offset
onClose(WsSession session, int statusCode, String r
onError(WsSession session, Throwable throwable)
```

The `WsSession` object contains methods for getting path-params and query-params, as well as methods for sending data to the client.

Configuring the server. Javalin doesn't require an application to run, because it runs on top of an embedded Jetty server. Javalin provides several helpful configuration options, all of which are programmatic (there are no configuration files). The following snippet shows some of the options:

```
Javalin.create()
    .contextPath("/context-path")
    .enableAutogeneratedEtags()
    .enableCorsForOrigin("*")
    .enableDebugLogging()
    .enableStaticFiles("/public")
    .start();
```

If you need more control than what Javalin exposes through its configuration API, you can supply Javalin with your own Jetty `Server` object:

```
app.server(() -> {
    Server server = new Server(); //org.eclipse.jet
    // configure server
    return server;
});
```

You can use this option if you want to run Javalin on an HTTP/2 server. (The code required to set up Jetty to run with HTTP/2 is too long to include in this article, but there is a working example on GitHub.)

The process is similar for configuring a Jetty `SessionHandler`, and an extensive tutorial is available on the Javalin website.

Advanced Concepts

Handler groups. When you build a larger application, you often end up with routes that share the same path. For example, consider a standard CRUD API for users:

```
app.get("/users/", UserController::getAll)
app.post("/users/", UserController::create)
app.get("/users/:user-id", UserController::getOne)
app.patch("/users/:user-id", UserController::update
app.delete("/users/:user-id", UserController::delet
```

To reduce the amount of noise in these kinds of apps, Javalin has the concept of handler groups, which define a block scope where the `app` object is the receiver and thereby allows you to write tighter code:

```
app.routes {
    path("users") {
        get(UserController::getAll)
        post(UserController::create)
        path(":user-id") {
            get(UserController::getOne)
            patch(UserController::update)
            delete(UserController::delete)
        }
    }
}
```

Handler groups improve readability and significantly reduce the potential for programming errors. Instead of repeating `users` five times and `:user-id` three times, each string is now used only once. This eliminates the need to extract strings into variables, leaving the code more readable and less error-prone.

Asynchronous responses. Asynchronous request handling is simple in Javalin. If you set the `Context` result to be a `CompletableFuture`, Javalin will remove the request from the thread pool and finish it asynchronously. This option improves performance by freeing up the thread pool to deal with new requests instead of waiting for database calls or HTTP requests to finish. Several libraries return `CompletableFuture` in Java, which makes things even simpler. Here is an example using Java 11 and jasync-sql, a database driver for MySQL and PostgreSQL:

```
app.get("/", ctx -> {
    var futureResult =
        connection.sendPreparedStatement("select 0"
                  .thenApply(...)
    ctx.result(futureResult);
});
```

As you can see, there is no difference between an async handler and a blocking one, except for the type of object you give to `ctx.result()`.

Access management. Most production applications eventually need some sort of access management. Security is not something for which Javalin is responsible, but the framework does give you the tools to easily create your own implementation. Every HTTP request in a Javalin application is run through an `AccessManager`. The default implementation is to allow every request, so developers are responsible for defining their own security. Consider the following snippet:

```
get("/secured", ctx -> ctx.result("!"), roles(MY_RO
```

Here, the code defines a GET handler for the path `"/secured"` and attaches the role `MY_ROLE` to it. So how do you get Javalin to respect this role? As with most other concepts in Javalin, `AccessManager` is a functional interface:

```
app.accessManager((handler, ctx, permittedRoles) ->
    handler.handle(ctx); // handle the request
});
```

The parameter permittedRoles is of type Set<role>, and it contains the roles attached to the endpoint. If you have not attached any roles to your endpoint, you can ignore it. If you do have roles attached to your endpoint, you can use it to determine whether the user should have access to the endpoint:

```
app.accessManager((handler, ctx, permittedRoles) ->
    if (permittedRoles.contains(getUserRole(ctx)) {
        handler.handle(ctx);
    } else {
        ctx.status(401);
    }
});
```

There are no predefined roles in Javalin. The recommended approach is to create an enum that implements Role, which is a marker interface—that is, an empty interface.

When to Use Javalin

Javalin is simple and unopinionated, which makes it a good choice if you need to get started quickly. Its abstraction layer is thin, which makes it easy to understand what's going on under the hood. Javalin also is fast, serving 1.1 million requests per second (rps) in the October 2018 TechEmpower benchmarks, which is significantly faster than most heavier frameworks and many lightweight frameworks.

Javalin works well with GraalVM (there's a tutorial on the website). The final binary is only 22 MB (everything included) and starts instantly.

The simplicity of Javalin comes at a cost. Because Javalin does only web applications, developers need to solve database setup, dependency injection, command-line parsing, and other important aspects of an application. The Javalin website has numerous tutorials that show how to approach many of these tasks.

Conclusion

This article presents just a quick overview of Javalin's functionality. As you can see, the scope of Javalin is narrow and limited to the web layer. The codebase is small, and tests make up the majority of it (6,000 out of 10,000 lines of code). If you're interested in contributing, please visit the project on GitHub. Otherwise, consider using Javalin for your projects, both commercial and personal, whenever you need a fast, lightweight web framework.

Also in This Issue

David Åse

David Åse is a software engineer at Working Group Two, a telecommunications startup. He graduated from the Norwegian University of Science and Technology with a master's degree in computer science in 2014 and immediately joined the open source project Spark Java. He is now an open source enthusiast and the creator and maintainer of two popular open source Java projects: Javalin and j2html.