Chapter One: Introduction to Javalin.

What is Javelin?

Javalin is a web framework runs on top of Jetty, one of the most used and stable web-servers on the JVM. You can configure the Jetty server fully, so you can easily get SSL and HTTP2 and everything else that Jetty has to offer. You can use Javelin with either `Java` or `Kotlin` languages

Javalin started as a fork of the Java and Kotlin web framework Spark, but quickly turned into a ground-up rewrite influenced by koa.js. Both of these web frameworks are inspired by the modern micro web framework grandfather: Sinatra, so if you're coming from Ruby then Javalin shouldn't feel *too* unfamiliar.

Philosophy

Like Sinatra, Javalin is not aiming to be a full web framework, but rather just a lightweight REST API library (or a micro framework, if you must). There is no concept of MVC, but there is support for template engines, WebSockets, and static file serving for convenience. This allows you to use Javalin for both creating your RESTful API backend, as well as serving an `index.html` with static resources (in case you're creating an SPA). This is practical if you don't want to deploy an apache or nginx server in addition to your Javalin service. If you wish to use Javalin to create a more traditional website instead of a REST APIs, there are several template engine wrappers available for a quick and easy setup.

API design

All the methods on the Javalin instance return `this`, making the API fully fluent. This will let you create a declarative and predictive REST

API, that will be very easy to reason about for new developers joining your project.

Java and Kotlin interoperability

Javalin is both a Kotlin web framework and a Java web framework, meaning the API is being developed with focus on great interoperability between the two languages. The library itself is written primarily in Kotlin, but has a few core classes written in Java to achieve the best interoperability between the two languages. Javalin is intended as a “foot in the door” to Kotlin development for companies that already write a lot of Java.

When moving a Javalin project from Java to Kotlin, you shouldn’t need to learn a new way of doing things. To maintain this consistent API for both languages is an important goal of the project.

Sailient features of Javalin

Simple

Unlike other Java and Kotlin web frameworks, Javalin has very few concepts that you need to learn. You never have to extend a class and you rarely have to implement an interface.

Lightweight

Javalin is just a few thousand lines of code on top of Jetty, which means its performance is almost equivalent to pure Jetty. It also means it’s very easy to reason about the source code.

Active

A new version of Javalin has been released twice a month (on average) since the first version. Don’t worry though, every version is backwards compatible. PRs and issues are reviewed swiftly, normally every week.

Interoperable

Other Java and Kotlin web frameworks usually offer separate version for each language. Javalin is being developed with interoperability in mind, so apps are built the same way in both Java and Kotlin.

Flexible

Javalin is designed to be simple and blocking, as this is the easiest programming model to reason about. However, if you set a Future as a result, Javalin switches into asynchronous mode.

Why should I use Javelin over Spring Boot or Vert.x?

Javalin is a microservice framework with a built in app server (making it easy to produce *fat jars*). Unlike Vert.x this has a simple programming model and is easier to code, debug and maintain. Spring Boot is "heavy" framework and typically requires a lot of setup for simple API app. The binaries produced are also fatter.

Javalin helps you write API based applications easily with minimal overhead and gets the job done. You also have full access to the power of JVM threading and asynchronous calls. It is, of course, possible to use all existing Java libraries, thus making Javalin an easy fit in your standard Java development ecosystem.

Can I write asynchronous code with Javalin?

Yes you can. Just set a `CompletableFuture<String>` or `CompletableFuture<InputStream>` as your result:

```
import io.javalin.Javalin

fun main(args: Array<String>) {
    val app = Javalin.start(7000)
    app.get("/") { ctx -> ctx.result(getFuture()) }
}

// hopefully your future is less pointless than this:
private fun getFuture() = CompletableFuture<String>().app\
ly {
    Executors.newSingleThreadScheduledExecutor().schedule\
({ this.complete("Hello World!") }, 1, TimeUnit.SECONDS)
}
```

How “Fast” is Javelin?

This really depends on your application. Javelin apps will be “as fast” as a standard Java application

How easy is Javelin to Learn?

Javalin is a micro-framework which really does not require any additional knowledge if you are already an existing Java developer who has developed a web app before. Please note that this is *not* an introductory book and there are several excellent online resources and books available which teach Java and Java Web Programming basics. It is recommended to familiarize oneself with these concepts before proceeding.

Summary

Javalin is a fascinating micro-framework. I really see it as a sort of NodeJS of the JVM. Something that is needed with the abundance of opinionated and heavy-weight frameworks out there.

Chapter Two : “Hello Javelin”

Setting up your IDE

Provide instructions and a screenshot to download IntelliJ Idea.

Creating a project

IntelliJ IDEA lets you create a Maven project or add a Maven support to any existing project.

- Launch the New Project wizard. If no project is currently opened in IntelliJ IDEA, click Create New Project on the Welcome screen: Otherwise, select File | New | Project from the main menu.

- Select Maven from the options on the left.
- Specify project's SDK (JDK) or use a default one and an archetype if you want to use a predefined project template (configure your own archetype by clicking Add Archetype).
- Click Next.
- On the next page of the wizard, specify the following Maven basic elements that are added to the pom.xml file:
    - GroupId - a package of a new project.
    - ArtifactId - a name of your project.
    - Version - a version of a new project. By default, this field is specified automatically.
- Click Next.
- If you are creating a project using a Maven archetype, IntelliJ IDEA displays Maven settings that you can use to set the Maven home directory and Maven repositories. Also, you can check the archetype properties. Click Next. Specify the name and location settings. Click Finish.

Setting up Maven

You should have the following code in your pom.xml:

Example 1: pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-inst\
ance"
         xsi:schemaLocation="http://maven.apache.org/POM/\
4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com</groupId>
    <artifactId>manish</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupI\
d>
                <artifactId>maven-compiler-plugin</artifa\
ctId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>io.javalin</groupId>
            <artifactId>javalin</artifactId>
            <version>1.7.0</version>
        </dependency>
    </dependencies>
</project>
```

TIP: To download source code of Javalin: `mvn dependency:sources`

A simple “Hello Javelin” app

Create a class called `HelloWorld` and write the following code in it:

Example 2: A Simple Hello World App

```java
import io.javalin.Javalin;
public class HelloWorld {
    public static void main(String[] args) {
        Javalin app = Javalin.start(7000);
        app.get("/", ctx -> ctx.result("Hello Javelin"));
    }
}
```

A quick overview of the “Hello Javelin” App

Let us now go through the application describing what each component does in detail.

```java
Javalin app = Javalin.start(7000);
```

The code above starts the application on port 7000. Let us see what internally happens when we start a Javalin app.

Javalin app internals: Startup

Example 3: Internals of Javalin startup

```java
public Javalin start() {
    if (!started) {
        if (!hideBanner) {
            log.info(Util.INSTANCE.javalinBanner());
        }
        Util.INSTANCE.printHelpfulMessageIfLoggerIsMissin\
g();
        Util.INSTANCE.setNoServerHasBeenStarted(false);
        eventManager.fireEvent(EventType.SERVER_STARTING,\
 this);
        try {
            embeddedServer = embeddedServerFactory.create\
(new JavalinServlet(
                contextPath,
                pathMatcher,
                exceptionMapper,
                errorMapper,
                jettyWsHandlers,
                javalinWsHandlers,
                logLevel,
                dynamicGzipEnabled,
                defaultContentType,
                defaultCharacterEncoding,
                maxRequestCacheBodySize
            ), staticFileConfig);
            log.info("Starting Javalin ...");
            port = embeddedServer.start(port);
            log.info("Javalin has started \\o/");
            started = true;
            eventManager.fireEvent(EventType.SERVER_START\
ED, this);
        } catch (Exception e) {
            log.error("Failed to start Javalin", e);
            if (e instanceof BindException && e.getMessag\
e() != null) {
                if (e.getMessage().toLowerCase().contains\
("in use")) {
                    log.error("Port already in use. Make \
sure no other process is using port " + port + " and try \
again.");
                } else if (e.getMessage().toLowerCase().c\
ontains("permission denied")) {
                    log.error("Port 1-1023 require elevat\
ed privileges (process must be started by admin).");
                }
            }
            eventManager.fireEvent(EventType.SERVER_START\
_FAILED, this);
        }
    }
    return this;
}
```

Javalin app internals:

The second line of the Javalin app reads:

```
app.get("/", ctx -> ctx.result("Hello Javelin"));
```

The Context instance.

The `ctx` represents an instance of the `Context` object, which contains everything you need to handle an `http` request. It contains the underlying servlet-request and servlet-response, and a bunch of getters and setters. The getters operate mostly on the request-object, while the setters operate exclusively on the response object.

Context methods: A reference

```
// REQUEST METHODS
ctx.request();
// get underlying HttpServletRequest
ctx.anyFormParamNull("k1", "k2");
// returns true if any form-param is null
ctx.anyQueryParamNull("k1", "k2");
// returns true if any query-param is null
ctx.body();
// get the request body as string
ctx.bodyAsBytes();
// get the request body as byte-array
ctx.bodyAsClass(clazz);
// convert json body to object
ctx.formParam("key");
```

```
// get form param
ctx.formParams("key");
// get form param with multiple values
ctx.formParamMap();
// get all form param key/values as map
ctx.param("key");
// get a path-parameter, ex "/:id" -> param("id")
ctx.paramMap();
// get all param key/values as map
ctx.splat(0);
// get splat by nr, ex "/*" -> splat(0)
ctx.splats();
// get array of splat-values
ctx.attribute("key", "value");
// set a request attribute
ctx.attribute("key");
// get a request attribute
ctx.attributeMap();
// get all attribute key/values as map
ctx.basicAuthCredentials()
// get username and password used for basic-auth
ctx.contentLength();
// get request content length
ctx.contentType();
// get request content type
ctx.cookie("key");
// get cookie by name
ctx.cookieMap();
// get all cookie key/values as map
ctx.header("key");
// get a header
ctx.headerMap();
// get all header key/values as map
ctx.host();
// get request host
ctx.ip();
// get request up
ctx.isMultipart();
// check if request is multipart
ctx.mapFormParams("k1", "k2");
// map form params to their values, returns null if any f\
orm param is missing
ctx.mapQueryParams("k1", "k2");
// map query params to their values, returns null if any \
query param is missing
ctx.matchedPath();
// get matched path, ex "/path/:param"
ctx.next();
// pass the request to the next handler
ctx.path();
// get request path
ctx.port();
// get request port
ctx.protocol();
// get request protocol
ctx.queryParam("key");
// get query param
ctx.queryParams("key");
// get query param with multiple values
ctx.queryParamMap();
// get all query param key/values as map
ctx.queryString();
// get request query string
ctx.method();
// get request method
ctx.scheme();
// get request scheme
ctx.sessionAttribute("foo", "bar");
// set session-attribute "foo" to "bar"
```

```
ctx.sessionAttribute("foo");
// get session-attribute "foo"
ctx.sessionAttributeMap();
// get all session attributes as map
ctx.uploadedFile("key");
// get file from multipart form
ctx.uploadedFiles("key");
// get files from multipart form
ctx.uri();
// get request uri
ctx.url();
// get request url
ctx.userAgent();
// get request user agent

// RESPONSE methods

ctx.response();
// get underlying HttpServletResponse
ctx.result("result");
// set result (string)
ctx.result(inputStream);
// set result (stream)
ctx.result(future);
// set result (future)
ctx.resultString();
// get response result (string)
ctx.resultStream();
// get response result (stream)
ctx.resultFuture();
// get response result (future)
ctx.charset("charset");
// set response character encoding
ctx.header("key", "value");
// set response header
ctx.html("body html");
// set result and html content type
ctx.json(object);
// set result with object-as-json
ctx.redirect("/location");
// redirect to location
ctx.redirect("/location", 302);
// redirect to location with code
ctx.status();
// get response status
ctx.status(404);
// set response status
ctx.cookie("key", "value");
// set cookie with key and value
ctx.cookie("key", "value", 0);
// set cookie with key, value, and maxage
ctx.cookie(cookieBuilder);
// set cookie using cookiebuilder
ctx.removeCookie("key");
// remove cookie by key
ctx.removeCookie("/path", "key");
// remove cookie by path and key
```

Server Startup and Lifecyle management

To start and stop the server, use the appropriately named `start()` and `stop()` methods.

Starting and Stopping server programmatically

```
Javalin app = Javalin.create()
```

```
    .start() // start server (sync/blocking)
    .stop() // stop server (sync/blocking)
```

Starting without any custom configuration

```
Javalin app = Javalin.start(7000);
```

Server setup: Available configurations

Server setup: Available configurations

```
Javalin.create() // create has to be called first
    .contextPath("/context-path")
    // set a context path (default is "/")
    .dontIgnoreTrailingSlashes()
    // treat '/test' and '/test/' as different URLs
    .defaultContentType(string)
    // set a default content-type for responses
    .defaultCharacterEncoding(string)
    // set a default character-encoding for responses
    .disableStartupBanner()
    // remove the javalin startup banner from logs
    .embeddedServer( ... )
    // see section below
    .enableCorsForOrigin("origin")
    // enables cors for the specified origin(s)
    .enableDynamicGzip()
    // gzip response (if client accepts gzip and response\
 is more than 1500 bytes)
    .enableRouteOverview("/path")
    // render a HTML page showing all mapped routes
    .enableStandardRequestLogging()
    // does requestLogLevel(LogLevel.STANDARD)
    .enableStaticFiles("/public")
    // enable static files (opt. second param Location.CL\
ASSPATH/Location.EXTERNAL)
    .maxBodySizeForRequestCache(long)
    // set max body size for request cache
    .port(port)
    // set the port
    .start();
    // start has to be called last
```

Custom Server

Starting a custom server

```
app.embeddedServer(new EmbeddedJettyFactory(() -> {
    Server server = new Server();
    // do whatever you want here
    return server;
}));
```

Custom jetty handlers

You can configure your embedded jetty-server with a handler-chain, and Javalin will attach it’s own handlers to the end of this chain.

Custom Jetty Hander example

```
StatisticsHandler statisticsHandler = new StatisticsHandl\
er();

Javalin.create()
```

```
    .embeddedServer(new EmbeddedJettyFactory(() -> {
        Server server = new Server();
        server.setHandler(statisticsHandler);
        return server;
    }))
    .start();
```

Implementing a custom server with SSL

Implementing a custom server with SSL enabled is easy in Javalin, but not straightforward. It may require hunting around in the documentation. For your reference, here is a complete working example with SSL

A SSL Hello World example

```
import io.javalin.Javalin;
import org.eclipse.jetty.server.Connector;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.ServerConnector;
import org.eclipse.jetty.util.ssl.SslContextFactory;

public class HelloWorldSecure {

    // This is a very basic example, a better one can be \
found at:
    // https://github.com/eclipse/jetty.project/blob/jett\
y-9.4.x/examples/embedded/src/main/java/org/eclipse/jetty\
/embedded/LikeJettyXml.java#L139-L163
    public static void main(String[] args) {
        Javalin.create()
            .server(() -> {
                Server server = new Server();
                ServerConnector sslConnector = new Server\
Connector(server, getSslContextFactory());
                sslConnector.setPort(443);
                ServerConnector connector = new ServerCon\
nector(server);
                connector.setPort(80);
                server.setConnectors(new Connector[]{sslC\
onnector, connector});
                return server;
            })
            .start()
            .get("/", ctx -> ctx.result("Hello World")); \
// valid endpoint for both connectors
    }

    private static SslContextFactory getSslContextFactory\
() {
        SslContextFactory sslContextFactory = new SslCont\
extFactory();
        sslContextFactory.setKeyStorePath(HelloWorldSecur\
e.class.getResource("/keystore.jks").toExternalForm());
        sslContextFactory.setKeyStorePassword("password");
        return sslContextFactory;
    }
}
```

Implementing a custom server with HTTP/2

The following is a sample server implemented with HTTP2. There is no straight-forward example easily, please use the following code below:

HTTP/2 server with Javalin

```java
import io.javalin.Javalin;
import io.javalin.embeddedserver.jetty.EmbeddedJettyFacto\
ry;
import org.eclipse.jetty.alpn.ALPN;
import org.eclipse.jetty.alpn.server.ALPNServerConnection\
Factory;
import org.eclipse.jetty.http2.HTTP2Cipher;
import org.eclipse.jetty.http2.server.HTTP2ServerConnecti\
onFactory;
import org.eclipse.jetty.server.HttpConfiguration;
import org.eclipse.jetty.server.HttpConnectionFactory;
import org.eclipse.jetty.server.SecureRequestCustomizer;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.server.ServerConnector;
import org.eclipse.jetty.server.SslConnectionFactory;
import org.eclipse.jetty.util.ssl.SslContextFactory;

public class Main {

    public static void main(String[] args) {

        Javalin app = Javalin.create()
            .embeddedServer(createHttp2Server())
            .enableStaticFiles("/public")
            .start();

        app.get("/", ctx -> ctx.result("Hello World"));

    }

    private static EmbeddedJettyFactory createHttp2Server\
() {
        return new EmbeddedJettyFactory(() -> {
            Server server = new Server();

            ServerConnector connector = new ServerConnect\
or(server);
            connector.setPort(8080);
            server.addConnector(connector);

            // HTTP Configuration
            HttpConfiguration httpConfig = new HttpConfig\
uration();
            httpConfig.setSendServerVersion(false);
            httpConfig.setSecureScheme("https");
            httpConfig.setSecurePort(8443);

            // SSL Context Factory for HTTPS and HTTP/2
            SslContextFactory sslContextFactory = new Ssl\
ContextFactory();
            sslContextFactory.setKeyStorePath(Main.class.\
getResource("/keystore.jks").toExternalForm());
            // replace with your real keystore
            sslContextFactory.setKeyStorePassword("passwo\
rd");
            // replace with your real password
            sslContextFactory.setCipherComparator(HTTP2Ci\
pher.COMPARATOR);
            sslContextFactory.setProvider("Conscrypt");

            // HTTPS Configuration
            HttpConfiguration httpsConfig = new HttpConfi\
guration(httpConfig);
```

```
            httpsConfig.addCustomizer(new SecureRequestCu\
stomizer());

            // HTTP/2 Connection Factory
            HTTP2ServerConnectionFactory h2 = new HTTP2Se\
rverConnectionFactory(httpsConfig);
            ALPNServerConnectionFactory alpn = new ALPNSe\
rverConnectionFactory();
            alpn.setDefaultProtocol("h2");

            // SSL Connection Factory
            SslConnectionFactory ssl = new SslConnectionF\
actory(sslContextFactory, alpn.getProtocol());

            // HTTP/2 Connector
            ServerConnector http2Connector = new ServerCo\
nnector(server, ssl, alpn, h2, new HttpConnectionFactory(\
httpsConfig));
            http2Connector.setPort(8443);
            server.addConnector(http2Connector);

            return server;
        });
    }
}
```

Note that you will have to generate a keystore locally using: `keytool -genkey -alias mydomain -keyalg RSA -keystore keystore.jks -keysize 2048`

Context Extensions:

Context extensions give Java developers a way of extending the Context object.

One of the most popular features of Kotlin is extension functions. When working with an object you don't own in Java, you often end up making `MyUtil.action(object, ...)`. If you, for example, want to serialize an object and set it as the result on the Context, you might do:

```
app.get("/", ctx -> MyMapperUtil.serialize(ctx, myMapper,\
 myObject));
```

With context extensions you can add custom extensions on the context:

```
app.get("/", ctx -> ctx.use(MyMapper.class).serialize(obj\
ect)); // use MyMapper to serialize object
```

Context extensions have to be added before you can use them, this would typically be done in the first before filter of your app:

```
app.before(ctx -> ctx.register(MyMapper.class, new MyMapp\
er(ctx, otherDependency));
```

AccessManager for Authentication and Authorization

Javalin has a functional interface `AccessManager`, which let's you set per-endpoint authentication and/or authorization. It's common to use before-handlers for this, but per-endpoint security handlers give you much more explicit and readable code. You can implement your access-manager however you want, but here is an example implementation:

Access manager code snippet

```java
// Set the access-manager that Javalin should use
app.accessManager((handler, ctx, permittedRoles) -> {
    MyRole userRole = getUserRole(ctx);
    if (permittedRoles.contains(userRole)) {
        handler.handle(ctx);
    } else {
        ctx.status(401).result("Unauthorized");
    }
});

Role getUserRole(Context ctx) {
    // determine user role based on request
    // typically done by inspecting headers
}

enum MyRole implements Role {
    ANYONE, ROLE_ONE, ROLE_TWO, ROLE_THREE;
}

app.routes(() -> {
    get("/un-secured",
        ctx -> ctx.result("Hello"), roles(ANYONE));
    get("/secured",
        ctx -> ctx.result("Hello"), roles(ROLE_ONE));
});
```

Exception Mapping

All handlers (before, endpoint, after) can throw `Exception` (and any subclass of `Exception`) The `app.exception()` method gives you a way of handling these exceptions:

```java
app.exception(NullPointerException.class, (e, ctx) -> {
    // handle nullpointers here
});

app.exception(Exception.class, (e, ctx) -> {
    // handle general exceptions here
    // will not trigger if more specific exception-mapper\
 found
});
```

Javalin has a `HaltException` which is handled before other exceptions. It can be used to short-circuit the request-lifecycle. If you throw a `HaltException` in a before-handler, no endpoint-handler will fire. When throwing a `HaltException` you can include a status code, a message, or both:

```
throw new HaltException();                         // (status\
: 200, message: "Execution halted")
throw new HaltException(401);                      // (status\
: 401, message: "Execution halted")
throw new HaltException("My message");             // (status\
: 200, message: "My message")
throw new HaltException(401, "Unauthorized");  // (status\
: 401, message: "Unauthorized")
```

Error Mapping

Error mapping is similar to exception mapping, but it operates on HTTP status codes instead of Exceptions:

```
app.error(404, ctx -> {
    ctx.result("Generic 404 message")
});
```

It can make sense to use them together:

```
app.exception(FileNotFoundException.class, (e, ctx) -> {
    ctx.status(404);
}).error(404, ctx -> {
    ctx.result("Generic 404 message")
});
```

WebSockets

Javalin has a very intuitive way of handling WebSockets, similar to most node frameworks:

```
app.ws("/websocket/:path", ws -> {
    ws.onConnect(session -> System.out.println("Connected\
"));
    ws.onMessage((session, message) -> {
        System.out.println("Received: " + message);
        session.getRemote().sendString("Echo: " + message\
);
    });
    ws.onClose((session, statusCode, reason) -> System.ou\
t.println("Closed"));
    ws.onError((session, throwable) -> System.out.println\
("Errored"));
});
```

The `WsSession` object wraps Jetty's `Session` and adds the following methods:

```
session.send("message")
// send a message to session remote (the ws client)
session.queryString()
// get query-string from upgrade-request
session.queryParam("key")
// get query-param from upgrade-request
session.queryParams("key")
// get query-params from upgrade-request
session.queryParamMap()
// get query-param-map from upgrade-request
session.mapQueryParams("k1", "k2")
// map query-params to values (only useful in kotlin)
session.anyQueryParamNull("k1", "k2")
// check if any query-param from upgrade-request is null
session.param("key")
// get a path-parameter, ex "/:id" -> param("id")
session.paramMap()
// get all param key/values as map
session.header("key")
// get a header
session.headerMap()
// get all header key/values as map
session.host()
// get request host
```

Lifecycle events

Javalin has five lifecycle events: `SERVER_STARTING`, `SERVER_STARTED`, `SERVER_START_FAILED`, `SERVER_STOPPING` and `SERVER_STOPPED`. The snippet below shows all of them in action:

```
Javalin app = Javalin.create()
    .event(EventType.SERVER_STARTING, e -> { ... })
    .event(EventType.SERVER_STARTED, e -> { ... })
    .event(EventType.SERVER_START_FAILED, e -> { ... })
    .event(EventType.SERVER_STOPPING, e -> { ... })
    .event(EventType.SERVER_STOPPED, e -> { ... });

app.start(); // SERVER_STARTING -> (SERVER_STARTED || SER\
VER_START_FAILED)
app.stop(); // SERVER_STOPPING -> SERVER_STOPPED
```

Adding a logger:

If you're reading this, you've probably seen the following message while running Javalin:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerB\
inder".
SLF4J: Defaulting to no-operation (NOP) logger implementa\
tion
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBi\
nder for further details.
```

Logger dependencies for slj4j in pom.xml

```xml
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.25</version>
</dependency>
```

Asynchronous requests

Javalin 1.6.0 introduced future results. While the default threadpool (200 threads) is enough for most use cases, sometimes slow operations should be run asynchronously. Luckily it's very easy in Javalin, just pass a `CompletableFuture` to `ctx.result()`:

Example of asynchronous requests

```kotlin
import io.javalin.Javalin

fun main(args: Array<String>) {
    val app = Javalin.start(7000)
    app.get("/") { ctx -> ctx.result(getFuture()) }
}

// hopefully your future is less pointless than this:
private fun getFuture() = CompletableFuture<String>().app\
ly {
    Executors.newSingleThreadScheduledExecutor()
     .schedule({ this.complete("Hello World!") }, 1, Time\
Unit.SECONDS)
}
```

You can only set future results in endpoint handlers (get/post/put/etc). After-handlers, exception-handlers and error-handlers run like you'd expect them to after the future has been resolved or rejected.

Configuring JSON Mapper and Jackson

The JSON mapper can be configured like this:

```java
Gson gson = new GsonBuilder().create();
JavalinJsonPlugin.setJsonToObjectMapper(gson::fromJson);
JavalinJsonPlugin.setObjectToJsonMapper(gson::toJson);
```

Configuring Jackson

The JSON mapper uses Jackson by default, which can be configured by calling:

```java
JavalinJacksonPlugin.configure(objectMapper)
```

Note that these are global settings, and can't be configured per instance of Javalin.

Views and Templates

Javalin currently supports five template engines, as well as markdown:

```
ctx.renderThymeleaf("/templateFile", model("firstName", "\
John", "lastName", "Doe"))
ctx.renderVelocity("/templateFile", model("firstName", "J\
ohn", "lastName", "Doe"))
ctx.renderFreemarker("/templateFile", model("firstName", \
"John", "lastName", "Doe"))
ctx.renderMustache("/templateFile", model("firstName", "J\
ohn", "lastName", "Doe"))
ctx.renderJtwig("/templateFile", model("firstName", "John\
", "lastName", "Doe"))
ctx.renderMarkdown("/markdownFile")
// Javalin looks for templates/markdown files in src/reso\
urces
```

Configure:

```
JavalinThymeleafPlugin.configure(templateEngine)
JavalinVelocityPlugin.configure(velocityEngine)
JavalinFreemarkerPlugin.configure(configuration)
JavalinMustachePlugin.configure(mustacheFactory)
JavalinJtwigPlugin.configure(configuration)
JavalinCommonmarkPlugin.configure(htmlRenderer, markdownP\
arser)
```

Note that these are global settings, and can’t be configured per instance of Javalin.

Chapter Three: Creating a Google Docs clone with WebSockets using Javalin

What You Will Learn

In this tutorial we will create a very simple realtime collaboration tool (like google docs). We will be using WebSockets for this, as WebSockets provides us with two-way communication over a one connection, meaning we won’t have to make additional HTTP requests to send and receive messages. A WebSocket connection stays open, greatly reducing latency (and complexity). Dependencies

Create a maven project with dependencies

We will be using Javalin for our web-server and WebSockets, and slf4j for logging:

pom.xml

```xml
<dependencies>
    <dependency>
        <groupId>io.javalin</groupId>
        <artifactId>javalin</artifactId>
        <version>1.7.0</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.25</version>
    </dependency>
</dependencies>
```

The Java application

The Java application is pretty straightforward. We need:

- a data class (Collab) containing the document and the collaborators
- a map to keep track of document-ids and Collabs
- websocket handlers for connect/message/close

We can get the entire server done in about 40 lines:

Websockets server example

```java
import io.javalin.Javalin;
import io.javalin.embeddedserver.jetty.websocket.WsSessio\
n;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class Main {

    private static Map<String, Collab> collabs = new Conc\
urrentHashMap<>();

    public static void main(String[] args) {

        Javalin.create()
            .port(7070)
            .enableStaticFiles("/public")
            .ws("/docs/:doc-id", ws -> {
                ws.onConnect(session -> {
                    if (getCollab(session) == null) {
                        createCollab(session);
                    }
                    getCollab(session).sessions.add(sessi\
on);
                    session.send(getCollab(session).doc);
                });
                ws.onMessage((session, message) -> {
                    getCollab(session).doc = message;
                    getCollab(session).sessions.stream().\
filter(WsSession::isOpen).forEach(s -> {
                        s.send(getCollab(session).doc);
                    });
                });
                ws.onClose((session, status, message) -> {
                    getCollab(session).sessions.remove(se\
ssion);
                });
            })
            .start();

    }

    private static Collab getCollab(WsSession session) {
        return collabs.get(session.param("doc-id"));
    }

    private static void createCollab(WsSession session) {
        collabs.put(session.param("doc-id"), new Collab()\
);
    }

}
```

We also need to create a data object for holding our document and the people working on it:

The Collboration object

```java
import io.javalin.embeddedserver.jetty.websocket.WsSessio\
n;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

public class Collab {
    public String doc;
    public Set<WsSession> sessions;

    public Collab() {
        this.doc = "";
        this.sessions = ConcurrentHashMap.newKeySet();
    }
}
```

Building a JavaScript Client

In order to demonstrate that our application works, we can build a JavaScript client. We'll keep the HTML very simple, we just need a heading and a text area:

```html
<body>
    <h1>Open the URL in another tab to start collaboratin\
g</h1>
    <textarea placeholder="Type something ..."></textarea>
</body>
```

The JavaScript part could also be very simple, but we want some slightly advanced features:

- When you open the page, the app should either connect to an existing document or generate a new document with a random id
- When a WebSocket connection is closed, it should immediately be reestablished
- When new text is received, the user caret (“text-cursor”) should remain in the same location (easily the most complicated part of the tutorial).

```
window.onload = setupWebSocket;
window.onhashchange = setupWebSocket;

if (!window.location.hash) { // document-id not present i\
n url
    const newDocumentId = Date.now().toString(36); // thi\
s should be more random
    window.history.pushState(null, null, "#" + newDocumen\
tId);
}

function setupWebSocket() {
    const textArea = document.querySelector("textarea");
    const ws = new WebSocket(`ws://localhost:7070/docs/${\
window.location.hash.substr(1)}`);
    textArea.onkeyup = () => ws.send(textArea.value);
    ws.onmessage = msg => {
    // place the caret in the correct position
        const offset = msg.data.length - textArea.value.l\
ength;
        const selection = {start: textArea.selectionStart\
, end: textArea.selectionEnd};
        const startsSame = msg.data.startsWith(textArea.v\
alue.substring(0, selection.end));
        const endsSame = msg.data.endsWith(textArea.value\
.substring(selection.start));
        textArea.value = msg.data;
        if (startsSame && !endsSame) {
            textArea.setSelectionRange(selection.start, s\
election.end);
        } else if (!startsSame && endsSame) {
            textArea.setSelectionRange(selection.start + \
offset, selection.end + offset);
        } else { // this is what google docs does...
            textArea.setSelectionRange(selection.start, s\
election.end + offset);
        }
    };
    ws.onclose = setupWebSocket; // should reconnect if c\
onnection is closed
}
```

And that’s it! Now try opening `localhost:7070` in a couple of different browser windows (that you can see simultaneously) and collaborate with yourself. Conclusion

We have a working realtime collaboration app written in less than 100 lines of Java and JavaScript. It’s very basic though, some things to add could include:

- Show who is currently editing the document
- Persist the data in a database at periodic intervals

- Replace the textarea with a rich text editor, such as quill
- Replace the textarea with a code editor such as ace for collaborative programming
- Improving the collaborative aspects with operational transformation

The use cases are not limited to text and documents though, you should use WebSockets for any project which requires a lot of interactions with low latency. Have fun!

Chapter Four: Creating a simple chat-app with WebSockets

Aim

In this tutorial we will create a simple real-time chat application. It will feature a chat-panel that stores messages received after you join, a list of currently connected users, and an input field to send messages from. We will be using WebSockets for this, as WebSockets provides us with full-duplex communication channels over a single TCP connection, meaning we won’t have to make additional HTTP requests to send and receive messages. A WebSocket connection stays open, greatly reducing latency (and complexity). Dependencies

First, we need to create a Maven project with some dependencies: (→ Tutorial)

pom.xml
```xml
<dependencies>
    <dependency>
        <groupId>io.javalin</groupId>
        <artifactId>javalin</artifactId>
        <version>1.7.0</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.13</version>
    </dependency>
    <dependency>
        <groupId>org.json</groupId>
        <artifactId>json</artifactId>
        <version>20160810</version>
    </dependency>
    <dependency>
        <groupId>com.j2html</groupId>
        <artifactId>j2html</artifactId>
        <version>1.2.0</version>
    </dependency>
</dependencies>
```

The Java application

The Java application is pretty straightforward. We need:

- a map to keep track of session/username pairs.
- a counter for number of users (nicknames are auto-incremented)
- websocket handlers for connect/message/close
- a method for broadcasting a message to all users
- a method for creating the message in HTML (or JSON if you prefer)

Websockets server example

```
public class Chat {

    private static Map<Session, String> userUsernameMap =\
 new ConcurrentHashMap<>();
    private static int nextUserNumber = 1; // Assign to u\
sername for next connecting user

    public static void main(String[] args) {
        Javalin.create()
            .port(7070)
            .enableStaticFiles("/public")
            .ws("/chat", ws -> {
                ws.onConnect(session -> {
                    String username = "User" + nextUserNu\
mber++;
                    userUsernameMap.put(session, username\
);
                    broadcastMessage("Server", (username \
+ " joined the chat"));
                });
                ws.onClose((session, status, message) -> {
                    String username = userUsernameMap.get\
(session);
                    userUsernameMap.remove(session);
                    broadcastMessage("Server", (username \
+ " left the chat"));
                });
                ws.onMessage((session, message) -> {
                    broadcastMessage(userUsernameMap.get(\
session), message);
                });
            })
            .start();
    }

    // Sends a message from one user to all users, along \
with a list of current usernames
    private static void broadcastMessage(String sender, S\
tring message) {
        userUsernameMap.keySet().stream().filter(Session:\
:isOpen).forEach(session -> {
            try {
                session.getRemote().sendString(
                    new JSONObject()
                        .put("userMessage", createHtmlMes\
sageFromSender(sender, message))
                        .put("userlist", userUsernameMap.\
values()).toString()
                );
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }

    // Builds a HTML element with a sender-name, a messag\
e, and a timestamp
    private static String createHtmlMessageFromSender(Str\
ing sender, String message) {
        return article(
            b(sender + " says:"),
            span(attrs(".timestamp"), new SimpleDateForma\
t("HH:mm:ss").format(new Date())),
            p(message)
        ).render();
    }

}
```

Building a JavaScript Client

In order to demonstrate that our application works, we can build a JavaScript client. First we create our index.html:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width, in\
itial-scale=1">
    <title>WebsSockets</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <div id="chatControls">
        <input id="message" placeholder="Type your messag\
e">
        <button id="send">Send</button>
    </div>
    <ul id="userlist"> <!-- Built by JS --> </ul>
    <div id="chat">    <!-- Built by JS --> </div>
    <script src="websocketDemo.js"></script>
</body>
</html>
```

As you can see, we reference a stylesheet called style.css, which can be found on GitHub.

The final step needed for completing our chat application is creating websocketDemo.js:

```
// small helper function for selecting element by id
let id = id => document.getElementById(id);

//Establish the WebSocket connection and set up event han\
dlers
let ws = new WebSocket("ws://" + location.hostname + ":" \
+ location.port + "/chat");
ws.onmessage = msg => updateChat(msg);
ws.onclose = () => alert("WebSocket connection closed");
```

```

// Add event listeners to button and input field
id("send").addEventListener("click", () => sendAndClear(i\
d("message").value));
id("message").addEventListener("keypress", function (e) {
    if (e.keyCode === 13) { // Send message if enter is p\
ressed in input field
        sendAndClear(e.target.value);
    }
});

function sendAndClear(message) {
    if (message !== "") {
        ws.send(message);
        id("message").value = "";
    }
}

function updateChat(msg) { // Update chat-panel and list \
of connected users
    let data = JSON.parse(msg.data);
    id("chat").insertAdjacentHTML("afterbegin", data.user\
Message);
    id("userlist").innerHTML = data.userlist.map(user => \
"<li>" + user + "</li>").join("");
}
```

And that’s it! Now try opening localhost:7070 in a couple of different browser windows (that you can see simultaneously) and talk to yourself.

Conclusion

Well, that was easy! We have a working real-time chat application implemented without polling, written in a total of less than 100 lines of Java and JavaScript. The implementation is very basic though, and we should at least split up the sending of the userlist and the messages (so that we don’t rebuild the user list every time anyone sends a message), but since the focus of this tutorial was supposed to be on WebSockets, I chose to do the implementation as minimal as I could be comfortable with.

Chapter Five: Working with HTML forms and a Java backend

Dependencies

First, we need to create a project with these dependencies: (→ Tutorial) pom.xml

```
<dependencies>
    <dependency>
        <groupId>io.javalin</groupId>
        <artifactId>javalin</artifactId>
        <version>1.7.0</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.13</version>
    </dependency>
</dependencies>
```

Setting up the backend

Create a Java file, for example Main.java, that has the following code:

Websockets server example

```
import java.util.HashMap;
import java.util.Map;

import io.javalin.Javalin;

public class Main {

    static Map<String, String> reservations = new HashMap\
<String, String>() {{
        put("saturday", "No reservation");
        put("sunday", "No reservation");
    }};

    public static void main(String[] args) {

        Javalin app = Javalin.create()
            .port(7777)
            .enableStaticFiles("/public")
            .start();

        app.post("/make-reservation", ctx -> {
            reservations.put(ctx.formParam("day"), ctx.fo\
rmParam("time"));
            ctx.html("Your reservation has been saved");
        });

        app.get("/check-reservation", ctx -> {
            ctx.html(reservations.get(ctx.queryParam("day\
")));
        });

    }

}
```

This will create an app which listens on port 7777, and looks for static files in your /src/resources/public folder. We have two endpoints mapped, one post, which will make a reservation, and one get, which will check your reservation. Settings up the HTML forms

Now we have to make two HTML forms for interacting with these endpoints. We can put these forms in a file

/resources/public/index.html, which will be available at http://localhost:7777/. Make reservation form

```html
<h2>Make reservation:</h2>
<form method="post" action="/make-reservation">
    Choose day
    <select name="day">
        <option value="saturday">Saturday</option>
        <option value="sunday">Sunday</option>
    </select>
    <br>
    Choose time
    <select name="time">
        <option value="8:00 PM">8:00 PM</option>
        <option value="9:00 PM">9:00 PM</option>
    </select>
    <br>
    <button>Submit</button>
</form>
```

To make a reservation we need to create something on the server (in this case it’s a simple map.put(), but usually you’d have a database). When creating something on the server, you should use the POST method, which can be specified by adding method=”post” to the <form> element.

In our Java code, we have a post endpoint: app.post(“/make-reservation”, ctx -> {…}. We need to tell our form to use this endpoint with the action attribute: action=”/make-reservation”. Actions are relative, so when you click submit, the browser will create a POST request to http://localhost:7777/make-reservation with the day/time values as the request-body. Check reservation form

```html
<h2>Check your reservation:</h2>
<form method="get" action="/check-reservation">
    Choose day
    <select name="day">
        <option value="saturday">Saturday</option>
        <option value="sunday">Sunday</option>
    </select>
    <br>
    <button>Submit</button>
</form>
```

To check a reservation we need to tell the server which day we’re interested in. In this case we’re not creating anything, and our action does not change the state of the server in any way, which makes it a good candidate for a GET request.

GET requests don’t have a request-body so when you click submit the browser creates a GET request to http://localhost:7777/check-reservation?day=saturday. The values of the form are added to the URL as query-parameters. HTML form GET vs POST summary

- POST requests should be used if the request can change the server state.
- POST requests have their information stored in the request-body. In order to extract information from this body you have to use ctx.formParam(key) in Javalin.
- Performing a series of GET requests should always return the same result (if no other POST request was performed in-between).
- GET requests have no request-body, and form information is sent as query-parameters in the URL. In order to extract information from this body you have to use ctx.queryParam(key) in Javalin.

File upload example

Let's expand our example a bit to include file uploads. We need to add a new dependency, a new endpoint, and a new form. Dependency

We need to add a dependency for handling file-uploads:

pom.xml
```
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
```

Endpoint

Upload example
```
app.post("/upload-example", ctx -> {
    ctx.uploadedFiles("files").forEach(file -> {
        try {
            FileUtils.copyInputStreamToFile(file.getConte\
nt(), new File("upload/" + file.getName()));
            ctx.html("Upload successful");
        } catch (IOException e) {
            ctx.html("Upload failed");
        }
    });
});
```

ctx.uploadedFiles("files") gives us a list of files matching the name files. We then save these files to an upload folder.

HTML form

```
<h1>Upload example</h1>
<form method="post" action="/upload-example" enctype="mul\
tipart/form-data">
    <input type="file" name="files" multiple>
    <button>Submit</button>
</form>
```

When uploading files you need to add enctype=”multipart/form-data” to your <form>. If you want to upload multiple files, add the multiple attribute to your <input>.

Chapter Six: A comparison with Spark

<h1 class=”no-margin-top”>SparkJava and Javalin comparison</h1> People often ask about the differences between Spark and Javalin. This page shows some of them. It’s not intended to tell you which library is better, just to highlight the differences.

Handlers vs Routes and Filters

Javalin has the concept of a `Handler`. A `Handler` is void and takes a `Context` (which wraps `HttpServletRequest` and `HttpServletResponse`), and you operate on both the request and response through this `Context`.

```
app.get("/path", ctx -> ctx.result("Hello, World!"));
app.after("/path", ctx -> ctx.result("Actually, nevermind\
...")); 
```

Spark on the other hand has `Route`s and `Filter`s. Both `Route` and `Filter` in Spark take (`Request`, `Response`) as input. `Route` has a return value (`Object`), while `Filter` is void.

`Request` and `Response` wrap `HttpServletRequest` and `HttpServletResponse`, respectively.

```
app.get("/path", (req, res) -> "Hello, World!");
app.after("/path", (req, res) -> res.body("Actually, neve\
rmind..."));
```

Some other differences:

- Spark can run on an application server. Javalin can only run on the included embedded Jetty server.

- Javalin supports a lot of configuration options (HTTP2, Jetty Handler chains, fully customizable Jetty `Server`, CORS, default values). Spark has sane defaults, but limited configuration options.
- Spark has a static API and an instance API. Javalin only has an instance API.
- Javalin focuses on Java/Kotlin interoperability and behaves the same in both languages. Spark has Kotlin DSL (spark-kotlin) built on top of itself with “Kotlin only” features.
- Spark supports route mapping based on accepts/content type. Javalin doesn’t.
- Javalin supports regex routes. Spark doesn’t.
- Javalin has a lambda based WebSocket API which supports path-params (and other things). Spark uses an annotation based API which proxies directly to Jetty.
- Spark has a redirect DSL `redirect.get("/fromPath", "/toPath");`. Javalin doesn’t.
- Javalin has async support for long running tasks via `CompletableFuture`. Spark doesn’t.
- Javalin supports context extensions (`ctx.extension(MyExt.class).myMethod()`). Spark doesn’t.
- Spark has a pac4j library. Javalin doesn’t.
- Javalin has an `AccessManager` interfaces with role-support. Spark doesn’t.
- Spark is written in Java. Javalin is written in Kotlin.
- Javalin has lifecycle events. Spark doesn’t.

Chapter 7: Javalin and Docker

What is Docker?

```
    Docker is a tool designed to make it easier to create\
, deploy, and run applications by using containers.
    Containers allow a developer to package up an applica\
tion with all of the parts it needs,
    such as libraries and other dependencies, and ship it\
 all out as one package.
    By doing so, thanks to the container, the developer c\
an rest assured that the application will run on any othe\
r
    Linux machine regardless of any customized settings t\
hat machine might have that could differ from the machine
    used for writing and testing the code.
    In a way, Docker is a bit like a virtual machine. But\
 unlike a virtual machine, rather than creating a whole
    virtual operating system, Docker allows applications \
to use the same Linux kernel as the system that they're
    running on and only requires applications be shipped \
with things not already running on the host computer.
    This gives a significant performance boost and reduce\
s the size of the application.
    &mdash; <a href="https://opensource.com/resources/wha\
t-docker">opensource.com</a>
```

Initial Setup

Before we get started, there are a few things we need to do:

- Set up Docker (Install Docker)
- Deploy Registry server (Deploy Registry)
- Install Maven
- Set up the Javalin Hello World example with Maven (→ Tutorial)

Configuring Maven

This is actually where most of the work is done. In order to easily deploy a Java application anywhere, you have to create a jar file containing your application and all of its dependencies. Open the pom.xml of your Javalin Maven project and add the following configuration (below your dependencies tag):

pom.xml

```
<distributionManagement>
        <repository>
            <id>repo</id>
            <name>Internal release</name>
            <url>your-snapshot-repo-url</url>
        </repository>
</distributionManagement>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.3.2</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.1.0</version>
            <executions>
                <!-- Run shade goal on package phase -->
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <shadedArtifactAttached>true</sha\
dedArtifactAttached>
                        <transformers>
                            <!-- add Main-Class to manife\
st file -->
```

```
                            <transformer
                                    implementation="org.a\
pache.maven.plugins.shade.resource.ManifestResourceTransf\
ormer">
                                <mainClass>HelloWorld</ma\
inClass>
                            </transformer>
                        </transformers>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Configuring Docker

Before we can configure anything we must create a Dockerfile. We can create a text file using any editor and name it Dockerfile. Copy below contents to the Dockerfile and move this file to root of your project.

```
FROM openjdk:8-jre-alpine

EXPOSE 7000
ENTRYPOINT ["/usr/bin/java", "-jar", "/usr/share/javalin/\
my-javalin.jar"]

ARG JAR_FILE
ADD target/${JAR_FILE} /usr/share/javalin/my-javalin.jar
```

When you’ve added the Dockerfile to your project, it should look like this

To build a docker image for your application we will use dockerfile-maven-plugin (dockerfile-maven-plugin). Set DOCKER_HOST environment variable as mentioned (here). In the repository section “localhost:5000” is the registry url.

pom.xml
```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin</artifactId>
    <version>1.3.6</version>
    <executions>
        <execution>
            <id>default</id>
            <goals>
                <goal>build</goal>
                <goal>push</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <repository>localhost:5000/javalin</repository>
        <tag>${project.version}</tag>
        <buildArgs>
```

```
      <JAR_FILE>${project.build.finalName}-shaded.j\
ar</JAR_FILE>
    </buildArgs>
  </configuration>
</plugin>
```

When you've added the Docker config to your pom, it should look like this

Making Javalin Listen on the Correct Port

The only thing left is making sure Javalin can handle your requests. We have exposed port 7000 in Dockerfile. That means that 7000 port on the container is accessible to the outside world. So we will configure Javalin to listen on "7000"

```java
import io.javalin.Javalin;

public class HelloWorld {

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

}
```

## Build and Push Docker image Now we can deploy our application using `mvn deploy`. This will build the docker image and push it to your registry server. Image name is same as repository value in the pom. Additionally we add a tag to image to specify images for different versions. So image name for this example is localhost:5000/javalin:1.0.0-SNAPSHOT. Again, make sure you're in your project root, then enter:

```
mvn deploy
```

Run Docker image

Now we can run our application using `docker run`. open terminal then enter:

```
docker run -d -p 7000:7000 localhost:5000/javalin:1.0.0-S\
NAPSHOT
```

That's it. Our application is now available at http://localhost:7000/

Chapter 8: Sending email with Javalin

Dependencies

First, we need to create a Maven project with some dependencies: (→ Tutorial)

pom.xml
```
<dependencies>
    <dependency>
        <groupId>io.javalin</groupId>
        <artifactId>javalin</artifactId> <!-- For handlin\
g http-requests -->
        <version>{{site.javalinversion}}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-email</artifactId> <!-- For s\
ending emails -->
        <version>1.4</version>
    </dependency>
    <dependency>
        <groupId>com.j2html</groupId>
        <artifactId>j2html</artifactId> <!-- For creating\
 HTML form -->
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId> <!-- For lo\
gging -->
        <version>1.7.25</version>
    </dependency>
</dependencies>
```

Setting up the backend

We need three endpoints: `GET '/'`, `POST '/contact-us'` and `GET '/contact-us/success'`:

example
```
import org.apache.commons.mail.*;
import io.javalin.Javalin;
import static j2html.TagCreator.*;

public class Main {

    public static void main(String[] args) {

        Javalin app = Javalin.create()
            .port(7000)
            .start();

        app.get("/", ctx -> ctx.html(
            form().withAction("/contact-us").withMethod("\
post").with(
                input().withName("subject").withPlacehold\
er("Subject"),
                br(),
                textarea().withName("message").withPlaceh\
older("Your message ..."),
                br(),
                button("Submit")
            ).render()
        ));

        app.post("/contact-us", ctx -> {
            Email email = new SimpleEmail();
            email.setHostName("smtp.googlemail.com");
            email.setSmtpPort(465);
            email.setAuthenticator(new DefaultAuthenticat\
or("YOUR_EMAIL", "YOUR_PASSWORD"));
            email.setSSLOnConnect(true);
            email.setFrom("YOUR_EMAIL");
            email.setSubject(ctx.formParam("subject")); /\
/ subject from HTML-form
            email.setMsg(ctx.formParam("message")); // me\
ssage from HTML-form
            email.addTo("RECEIVING_ADDRESS");
            email.send(); // will throw email-exception i\
f something is wrong
            ctx.redirect("/contact-us/success");
        });

        app.get("/contact-us/success", ctx -> ctx.html("Y\
our message was sent"));

    }

}
~~~~~~

In order to get the above code to work, you need to make \
some changes:

* Change `YOUR_EMAIL` to your gmail account <small>(youre\
mail@gmail.com)</small>
* Change `YOUR_PASSWORD` to your gmail password*
* Change `RECEIVING_ADDRESS` to where you want the email \
to be sent

**It might be a good idea to create a test-account instea\
d of using your real gmail credentials.*

When you have made the changes to the code, run the progr\
am and go to `http://localhost:7000`.
You will see a simple unstyled form with an input field, \
a textarea and a button.
Fill in the form and click the button to test your email \
server. After you click the button, your browser
is redirected to `/contact-us/success` (if the email was \
sent).

Any emails you have sent will show up in your `Sent` fold\
er in the gmail web-interface.

{bump-link-number}

{leanpub-filename="chapter9.txt"}

# Chapter 9: Deploying Javalin apps to Heroku

## What is Heroku?
        Heroku is a cloud application platform – a new wa\
y of building and deploying web apps.
        Our service lets app developers spend their time \
```

```
on their application code,
        not managing servers, deployment, ongoing operati\
ons, or scaling.
        &mdash; <a href="https://www.heroku.com/about">he\
roku.com</a>

Heroku takes care of everything related to deployment, an\
d gives
you easy access to key commands via their tool Heroku Too\
lbelt.
It's very easy to get started with (as you'll soon learn)\
, and it
provides a nice free-tier that you can use to deploy your\
 webapps.

## Initial Setup
Before we get started, there are a few things we need to \
do:

* Create a free Heroku account [(sign up)](https://signup\
.heroku.com/dc)
* Install [Heroku Toolbelt](https://toolbelt.heroku.com/)
* Install [Maven](https://maven.apache.org/guides/getting\
-started/maven-in-five-minutes.html)
* Set up the Javalin Hello World example with Maven [(→ T\
utorial)](/tutorials/maven-setup)

## Configuring Maven
This is actually where most of the work is done. In order\
 to easily
deploy a Java application anywhere, you have to create a \
jar file
containing your application and all of its dependencies.
Open the pom.xml of your Javalin Maven project and add the
following configuration (below your dependencies tag):

{title="pom.xml", lang=xml}
```

<build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-compiler-plugin</artifactId> <version>2.3.2</version> <configuration> <source>1.8</source> <target>1.8</target> </configuration> </plugin> <plugin> <artifactId>maven-assembly-plugin</artifactId> <executions> <execution> <phase>package</phase> <goals> <goal>single</goal> </goals> </execution> </executions> <configuration> <descriptorRefs> <descriptorRef>jar-with-dependencies</descriptorRef> </descriptorRefs> <archive> <manifest> <mainClass>Main</mainClass> </manifest> </archive> </configuration> </plugin> </plugins> </build>

Configuring Heroku

Before we can configure anything, we actually have to create a Heroku application. This can be done by using the `heroku`

`create` command.
Open a terminal and navigate to your project root, then enter:

```
heroku create javalin-heroku-example #choose your own app\
lication name
```

Now that you have a Heroku application, we have to configure how to deploy it using Maven. This is pretty straightforward using the Heroku Maven plugin.

We specify the JDK version and the app-name, along with the launch config:

pom.xml
```xml
<plugin>
    <groupId>com.heroku.sdk</groupId>
    <artifactId>heroku-maven-plugin</artifactId>
    <version>1.1.3</version>
    <configuration>
        <jdkVersion>1.8</jdkVersion>
        <appName>javalin-heroku-example</appName>
        <processTypes>
            <!-- Tell Heroku how to launch your applicati\
on -->
            <web>java -jar ./target/javalin-heroku-exampl\
e-1.0-jar-with-dependencies.jar</web>
        </processTypes>
    </configuration>
</plugin>
```

When you've added the Heroku config to your pom, it should look like this

Making Javalin Listen on the Correct Port

The only thing left is making sure Javalin can handle your requests. Heroku assigns your application a new port every time you deploy it, so we have to get this port and tell Javalin to use it:

```java
import io.javalin.Javalin;

public class Main {

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

    private static int getHerokuAssignedPort() {
        ProcessBuilder processBuilder = new ProcessBuilde\
r();
        if (processBuilder.environment().get("PORT") != n\
ull) {
            return Integer.parseInt(processBuilder.enviro\
nment().get("PORT"));
        }
        return 7000;
    }

}
```

Now we can deploy our application using `mvn heroku:deploy`.

Again, make sure you're in your project root, then enter:

```
mvn heroku:deploy
```

That's it. Our application is now available at `https://<appname>.herokuapp.com/`

Chapter 10: Authentication with Javalin

Dependencies

First, create a new Gradle project with the following dependencies: (→ Tutorial)

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jre8:$kot\
lin_version"
    compile "io.javalin:javalin:{{site.javalinversion}}"
    compile "com.fasterxml.jackson.module:jackson-module-\
kotlin:2.8.4"
    compile "org.slf4j:slf4j-simple:1.7.22"
}
```

## Creating controllers We need something worth protecting. Let's pretend we have a very important API for manipulating a user database. We make a controller-object with some dummy data and CRUD operations:

```
import io.javalin.Context
import java.util.*

object UserController {

    private data class User(val name: String, val email: \
String)

    private val users = hashMapOf(
            randomId() to User(name = "Alice", email = "a\
lice@alice.kt"),
                randomId() to User(name = "Bob", email = "bob\
@bob.kt"),
                randomId() to User(name = "Carol", email = "c\
arol@carol.kt"),
                randomId() to User(name = "Dave", email = "da\
ve@dave.kt")
    )

    fun getAllUserIds(ctx: Context) {
        ctx.json(users.keys)
    }

    fun createUser(ctx: Context) {
        users[randomId()] = ctx.bodyAsClass(User::class.j\
ava)
    }

    fun getUser(ctx: Context) {
        ctx.json(users[ctx.param(":user-id")!!]!!)
    }

    fun updateUser(ctx: Context) {
        users[ctx.param(":user-id")!!] = ctx.bodyAsClass(\
User::class.java)
    }

    fun deleteUser(ctx: Context) {
        users.remove(ctx.param(":user-id")!!)
    }

    private fun randomId() = UUID.randomUUID().toString()

}
```

We're using `!!` to convert nullables to non-nullables. If `:user-id` is missing or `users[id]` returns null, we'll get a NullPointerException and our application will crash. Handling this is outside the scope of the tutorial.

Creating roles

Now that we have our functionality, we need to define a set of roles for our system. This is done by implementing the `Role` interface from `io.javalin.security.Role`. We'll define three roles, one for "anyone", one for permission to read user-data, and one for permission to write user-data.

```
enum class ApiRole : Role { ANYONE, USER_READ, USER_WRITE\
 }
```

Setting up the API

Now that we have roles, we can implement our endpoints:

```
import io.javalin.ApiBuilder.*
import io.javalin.Javalin
import io.javalin.security.Role.roles

fun main(vararg args: String) {

    val app = Javalin.create().apply {
        accessManager(Auth::accessManager)
    }.start()

    app.routes {
        path("users") {
            get(UserController::getAllUserIds, roles(ApiR\
ole.ANYONE))
            post(UserController::createUser, roles(ApiRol\
e.USER_WRITE))
            path(":user-id") {
                get(UserController::getUser, roles(ApiRol\
e.USER_READ))
                patch(UserController::updateUser, roles(A\
piRole.USER_WRITE))
                delete(UserController::deleteUser, roles(\
ApiRole.USER_WRITE))
            }
        }
    }

}
```

A role has now been given to every endpoint:
* `ANYONE` can `getAllUserIds` * `USER_READ` can `getUser` * `USER_WRITE` can `createUser`, `updateUser` and `deleteUser`

Now, all that remains is to implement the access-manager (`Auth::accessManager`).

Implementing auth

The `AccessManager` interface in Javalin is pretty simple. It takes a `Handler` a `Context` and a list of `Role`s. The idea is that you implement code to run the handler based on what's in the context, and what roles are set for the endpoint.

The rules for our access manager are also simple: * When endpoint has `ApiRole.ANYONE`, all requests will be handled * When endpoint has another role set and the request has matching credentials, the request will be handled * Else we ignore the request and send `401 Unauthorized` back to the client

This translates nicely into Kotlin:

```
fun accessManager(handler: Handler, ctx: Context, permitt\
edRoles: List<Role>) {
    when {
        permittedRoles.contains(ApiRole.ANYONE) -> handle\
r.handle(ctx)
        ctx.userRoles.any { it in permittedRoles } -> han\
dler.handle(ctx)
        else -> ctx.status(401).json("Unauthorized")
    }
}
```

Extracting user-roles from the context

There is no `ctx.userRoles` concept in Javalin, so we need to implement it. First we need a user-table. We'll create a `map(Pair<String, String>, List<Role>)` where keys are username+password in cleartext (please don't do this for a real service), and values are user-roles:

```
private val userRoleMap = hashMapOf(
        Pair("alice", "weak-password") to listOf(ApiRole.\
USER_READ),
        Pair("bob", "better-password") to listOf(ApiRole.\
USER_READ, ApiRole.USER_WRITE)
)
```

Now that we have a user-table, we need to authenticate the requests. We do this by getting the username+password from the Basic-auth-header and using them as keys for the `userRoleMap`:

```
private val Context.userRoles: List<ApiRole>
    get() = this.basicAuthCredentials()?.let { (username,\
 password) ->
        userRoleMap[Pair(username, password)] ?: listOf()
    } ?: listOf()
```

When using basic auth, credentials are transferred as plain text (although base64-encoded). Remember to enable SSL if you're using basic-auth for a real service.**

Conclusion

This tutorial showed one possible way of implementing an `AccessManager` in Javalin, but the interface is very flexible and you can really do whatever you want:

```
app.accessManager(handler, ctx, permittedRoles) -> {
    when {
        ctx.host().contains("localhost") -> handler.handl\
e(ctx)
        Math.random() > 0.5 -> handler.handle(ctx)
        dayOfWeek == Calendar.SUNDAY -> handler.handle(ct\
x)
        else -> ctx.status(401).json("Unauthorized")
    }
};
```