zesty-router Documentation

Release 0.1.0

Stephen Maina

Mar 06, 2020

Contents

Zesty is a creative JavaScript wrapper around the Jetty web server. It works out-of-the-box with the servlet 3.0 API which enables handling requests in asynchronous fashion. Note that the servlet 3.1 API on the other hand, introduces new interfaces to handle asynchronous socket I/O for which there isn't an idiomatic way to handle generically in zesty yet. However, using either Javascript or Java with zesty is handled using a uniform and easy to understand API.

CHAPTER 1

AppServer Object

The is one of the core components in the zesty framework. It holds references to objects that play a central role in how everthing else is held together.:

```
public AppServer() {
    this(new HashMap<>());
}

public AppServer(Map<String, String> props) {
    this.assets(Optional.ofNullable(props.get("assets")).orElse("www"));
    this.appctx(Optional.ofNullable(props.get("appctx")).orElse("/"));
    this.engine(Optional.ofNullable(props.get("engine")).orElse("jtwig"));
    this.threadPoolExecutor = createThreadPoolExecutor();
}
```

1.1 locals: Properties

This holds configuration attributes that can be passed to the application and that are used by other components internally to configure their own behavior.

1.2 threadPoolExecutor: ThreadPoolExecutor

This is a thread-pool that is separate from the server's own thread pool. This is only used to handle requests for handlers configured as '*supports async*'. This thread pool is configurable using 3 parameters which are passed through the **locals** objects. These properties are:

- *poolSize* - the number of threads to keep in the pool, even if they are idle, unless allowCoreThreadTimeOut is set
- *maxPoolSize* - the maximum number of threads to allow in the pool

- *keepAliveTime* - when the number of threads is greater than the core, this is the maximum time (in MILLISECONDS) that excess idle threads will wait for new tasks before terminating.

1.3 wpcontext: Map

If configuring the application for `wordpress`, this will hold the configuration parameters required.:

```
// ************* WORDPRESS *****************//
public AppServer wordpress(String home, String fcgi_proxy) {
    this.wpcontext.put("activate", "true");
    this.wpcontext.put("resource_base", home);
    this.wpcontext.put("welcome_file", "index.php");
    this.wpcontext.put("fcgi_proxy", fcgi_proxy);
    this.wpcontext.put("script_root", home);
    return this;
}
```

1.4 corsconetxt: Map

This map holds the request headers' attributes that the application will use to configure the `CorsFilter`.

1.5 servlets: ServletContextHandler

This is the servlet handlers' container for the application. All configured routes are added to this handler under one context path.

1.6 engine: ViewEngine

This is an interface for exposing the active **view** component to the application. The value it holds specifies the concrete view implementation to be used by the application. This could be either be *freemarker*, *jtwig* at the moment.

CHAPTER 2

AppProvider Object

This is a provider for the **AppServer** object which is comes in handy when working with Javascript.:

```
public class AppProvider {

    public static AppServer provide(Map<String, String> props) {
        return new AppServer(props);
    }
}
```

You can alternatively just simply instantiate the **AppServer** manually.

CHAPTER 3

HandlerConfig Object

This interface allows you to dynamically customize the request handler (the underlying object is a `HttpServletRequest`) with the help of `ServletRegistration.Dynamic` to suit specific requirements.:

```
public interface HandlerConfig {

    public void configure(ServletHolder holder);
}
```

Through the `holder.getRegistration()` object, you will have access to servlet-specific methods like:

- setLoadOnStartup(int loadOnStartup)
- setMultipartConfig(MultipartConfigElement multipartConfig)
- Set<String> addMapping(String... urlPatterns)
- setAsyncSupported(boolean supported)

among others.

CHAPTER 4

BodyWriter Object

This interface allows you to take over the response generation responsibility and the output is sent back to the client as the response body.

```
public interface BodyWriter<T> {

    byte[] transform(T object);
}
```

CHAPTER 5

BodyReader Object

This interface allows you to take over the request body processing responsibility and the output is used by the application to process the request.

```
public interface BodyReader<T> {

    T transform(String type, byte[] bytes);
}
```

CHAPTER 6

Configure Logging

To configure logging, let's revisit how logback initialize itself.

- Logback tries to find a file called logback-test.xml in the classpath.
- If no such file is found, logback tries to find a file called logback.groovy in the classpath.
- If no such file is found, it checks for the file logback.xml in the classpath..
- If no such file is found, service-provider loading facility (introduced in JDK 1.6) is used to resolve the implementation of com.qos.logback.classic.spi.Configurator interface by looking up the file META-INFservicesch.qos.logback.classic.spi.Configurator in the class path. Its contents should specify the fully qualified class name of the desired Configurator implementation.
- If none of the above succeeds, logback configures itself automatically using the BasicConfigurator which will cause logging output to be directed to the console.

For the javascript application, we need another way to override this process in order to control logging. Fortunately, logback will allow us to configure a system property path which will preempt the initialization process above. `java -Dlogback.configurationFile=/path/to/config.xml`. To configure this for javascript, use the syntax below:

```
jjs --language=es6 -ot -scripting -J-Dlogback.configurationFile=../lib/app-logback.
↪xml -J-Djava.class.path=../lib/zesty-router-0.1.0-shaded.jar index.js
```

The corresponding `app-logback.xml` file would look something like this.:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">

    <appender name="STDOUT"
        class="ch.qos.logback.core.ConsoleAppender">
        <encoder
            class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <Pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </Pattern>
        </encoder>
```

(continues on next page)

(continued from previous page)

```
    </appender>

    <logger name="com.practicaldime.zesty" level="DEBUG" />
    <logger name="org.eclipse.jetty" level="ERROR" />

    <root level="debug">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

With this setup, you are back in control over the logging process through the configuration file.

CHAPTER 7

AppRoutes Object

This object holds the routes configured in the application to *route* requests to their respective handlers.:

```
public AppServer router() {
    this.routes = new AppRoutes(new MethodRouter());
    return this;
}
```

The AppRoutes object implements the Router which provides two methods - one for adding routes and the other for route lookup. It uses a tree structure and has a lookup efficiency equivalent to the height of the tree. The AppRoutes object itself holds the root of the tree. Each node in the tree implements the same Router interface.:

```
public interface Router {
    void accept(RouteSearch input);
    void addRoute(Route route);
}
```

The actual mapped routes are located in the leaf nodes, which makes the lookup time equivalent to the tree height. The nodes at each level are used to literally route the lookup down the correct path to the target route which matches the request. The root node is a method router - **GET, POST, PUT & DELETE**. It will drive the lookup down to the correct branch.:

```
public AppServer router() {
    this.routes = new AppRoutes(new MethodRouter());
    return this;
}
```

Each node accepts a RouteSearch objects which represents the incoming request, and uses this to determine which node it should pass the search object to.:

```
public void accept(RouteSearch input) {
    String method = input.requestAttrs.method;
    Method type = method != null? Method.valueOf(method.toUpperCase()) : null;
    if(type != null) {
        this.routers.get(type).accept(input);
```

(continues on next page)

(continued from previous page)

```
        //if a matching route is found, set the method value in the result
        if(input.requestAttrs != null) {
            input.result.method = type.name();
        }
    }
}
```

With a successful lookup, the `if(input.requestAttrs != null)` is not null, and each Router adds necessary information to the RouteSearch object as the search call stack unwinds. The method router node contains PathPartsRouter routers. These will route the lookup depending on the length of the path split along '/' (the path separator) character.:

```
public void accept(RouteSearch input) {
    String inputPath = input.requestAttrs.url;
    String path = inputPath != null? (inputPath.startsWith("/")? inputPath.
↪substring(1) : inputPath) : null;
    String[] parts = path != null? path.split("/") : null;
    if(parts != null) {
        Integer length = Integer.valueOf(parts.length);
        if(routers.containsKey(length)) {
            routers.get(length).accept(input);
        }
    }
}
```

The PathPartsRouter router node contains PathRegexRouter routers. These will route the lookup depending on the match found between the mapped routes and the request input represented by the RouteSearch object.:

```
public void accept(RouteSearch input) {
    for(PathPattern pathRegex : routers.keySet()) {
        Pattern valuesPattern = pathRegex.valuesPattern;
        Matcher matcher = valuesPattern.matcher(input.requestAttrs.url);
        if(matcher.matches()) {
            ...
        }
    }
}
```

If there is a match found, the request is routed to the HeadersRouter. The HeadersRouter is a leaf node. This will pull out the matching route based on the headers in the mapped route.:

```
public void accept(RouteSearch input) {
    List pool = new ArrayList<>(routes);
    for(Iterator iter = pool.iterator(); iter.hasNext();) {
        Route mapping = iter.next();
        //is 'content-type' declared in mapping route?
        if(mapping.contentType != null && mapping.contentType.trim().length() > 0) {
            ...
        }
        ...
        input.result = pool.size() > 0? pool.get(0) : null;
    }
}
```

At this point, the RouteSearch will either have the result property assigned a matched route of set to null, and the call stack begins to unwind back to the root node. The void addRoute(Route route); in each router is called by the

AppServer when accepting routes mapped by a user. Each path taken to place the route node in the correct place is the same path which is used to look up the node based on a request, which makes the logic simpler to understand.

CHAPTER 8

HandlerRequest Object

The HandlerRequest object is a wrapper around the HttpServletRequest object. Besides proving access to the entire HttpServletRequest API, the HandlerRequest object also adds some convenient methods which abstract the mundane and arduous tasks for performing some pretty common chores.

8.1 protocol() : String

Wraps around HttpServletRequest's getProtocol method

8.2 secure() : boolean

Returns true if the protocol is https

8.3 hostname() : String

Wraps around HttpServletRequest's getRemoteHost method

8.4 ip() : String

Wraps around HttpServletRequest's getRemoteAddr method

8.5 path() : String

Wraps around HttpServletRequest's getRequestURI method

8.6 route() : RouteSearch

Returns the route matched together with path parameters discovered as a Map<String, String> pathParams and the RequestAttrs used to search for the route.

8.7 param(name: String) : String

First searches for a named path parameter extracted from the request uri. This is defined using {curly braces} in the request path mapping. If none if found, it falls back to the servlet's `getParameter(name)` method which will return the named parameter if it exists in the request body. If none is found, it return s a null value.

8.8 pathParams() : Map

Returns a map of all named path parameters extracted from the request uri. These are defined using `{curly braces}` in the request path mapping.

8.9 query() : String

Wraps around HttpServletRequest's getQueryString method

8.10 header(name: String) : String

Wraps around HttpServletRequest's getHeader(name) method

8.11 error() : boolean

Return an error status associated with the request if something went wrong and was captured

8.12 message() : String

Return a message associated with the request if one was saved when the request was processed.

8.13 body() : byte[]

Return the request body captured if the long capture() was called prior.

8.14 body(type: Class) : T

Used to return the request's body() content as either json or xml depending on the respective Content-Type header in the request.

8.15 body(provider: BodyReader) : T

Used to return the request's body() content that is transformed using the BodyReader implementation supplied by caller of this method.

8.16 capture() : long

Captures the request body into a byte[] which is subsequently accessible using the byte[] body() method.

8.17 upload(destination: String) : long

Used together with a servlet designated for uploading multipart/form-data using the POST method and that is configured with a MultipartConfigElement object. The configured values for (String location, long maxFileSize, long maxRequestSize, int fileSizeThreshold) are ("temp", 1024 * 1024 * 50, 1024 * 1024, 5) respectively.:

```
router.get('/upload', function (req, res) {
    res.render('upload', {}); //return upload page
});

router.post('/upload', '', 'multipart/form-data', function (req, res) {
    var dest = req.param('destination');
    req.upload(dest);
    res.redirect(app.resolve("/upload")); //redirect to upload page
});
```

And the equivalent Java syntax would be.:

```
get("/upload", (HandlerRequest request, HandlerResponse response) -> {
    response.render("upload", Maps.newHashMap());
});

post("/www/upload", "", "multipart/form-data", (HandlerRequest request,␣
↪HandlerResponse response) ->{
    String dest = request.param("destination");
    request.upload(dest);
    response.redirect(app.resolve("/upload"));
});
```

8.18 cookies() : Cookie[]

Returns the cookies associated with the request.

8.19 session(create: boolean) : HttpSession

Returns the session object associated with the request. If the create attribute is true, a new session is created and returned. If it is false it will only return a session if one already exists otherwise it will not create one and returns nothing.

CHAPTER 9

HandlerResponse Object

The HandlerResponse object is a wrapper around the HttpServletResponse object. Besides proving access to the entire HttpServletResponse API, the HandlerResponse object also adds some convenient methods which abstract the mundane and arduous tasks for performing some pretty common chores.

9.1 header(header: String, value: String) : void

Wraps around HttpServletResponse's setHeader method

9.2 templates(folder: String) : void

Sets the directory name where the configured template engine will look up files

9.3 context(ctx: String): void

Sets the contextPath value in the request to that configured in the application level

9.4 status(status: int) : void

Wraps around HttpServletResponse's `setStatus(status)` method

9.5 sendStatus(status: int) : void

Set the response status through status(int status) and the sends the status message as the response body

9.6 end(payload: String) : void

Sends the payload in UTF-8 format as the response body. Example usage.:

```
let Date = Java.type('java.util.Date');
let DateFormat = Java.type('java.text.SimpleDateFormat');
function now() {
    return new DateFormat("hh:mm:ss a").format(new Date());
}
router.get('/', function (req, res) {
    res.send(app.status().concat(" @ ").concat(now()));
});
```

9.7 json(payload: Object) : void

Sends the payload in UTF-8 format and content-type as *application/json* as the response body

9.8 jsonp(payload: Object) : void

Sends the payload in UTF-8 format and content-type as *application/json* as the response body

9.9 xml(payload: Object, template: Class) : void

Sends the payload in UTF-8 format and content-type as *application/json* as the response body. The template variable is used to determine the structure of the xml payload

9.10 content(payload: T, writer: BodyWriter) : void

Sends the payload transformed by the writer object as the response body. The writer object is supplied by the caller of this method

9.11 render(template: String, model: Map) : void

Delegate the task of writing the response to the configured template engine. The engine will locate and load the template and merge it with the model object to create the response body

9.12 next(path: String) : void

Sets the forward flag to true, and sets the target uri to the path value prior to calling the request dispatcher to forward the request to the target resource.

9.13 redirect(path: String) : void

Sets the redirect flag to true, sets the target uri to the path value and sets the response status to 303 (see other) prior to calling the request to redirect to the target resource. Example usage.:

```
router.get('/target', function (req, res) {
    res.redirect(app.resolve("/dest/target"));
});
```

9.14 redirect(status: int, path: String) : void

Sets the redirect flag to true, sets the target uri to the path value and sets the response status to status prior to calling the request to redirect to the target resource. A redirect could be done using either 301, 302 or 302 depending on your context, so this redirect method gives you that discretion.

9.15 type(mimetype: String) : void

Wraps around HttpServletResponse's setContentType method

9.16 cookie(name: String, value: String) : void

Wraps around HttpServletResponse's addCookie method

9.17 attachment(filename: String) : void

Wraps around the download method and provides default values for attributes as shown: `download(filename, filename, getContentType(), null);`

9.18 download(path: String, filename: String, mimeType: String, status: HandlerStatus) : void

Reads the specified file relative the the application's root, and writes it to the response as a raw byte stream. An optional HandlerStatus callback interface is used to communicate progress as the file gets written

9.19 getContent() : byte[]

Return the byte[] content stored in the response object

CHAPTER 10

App Functions

These are the functions you can be able to use that are made accessible through the `AppServer` instance.:

```
let zesty = Java.type('com.practicaldime.router.base.AppProvider');

let Date = Java.type('java.util.Date');
let DateFormat = Java.type('java.text.SimpleDateFormat');

function now() {
    return new DateFormat("hh:mm:ss a").format(new Date());
}

//create AppServer instance
let app = zesty.provide({
    appctx: "/app",
    assets: "",
    engine: "freemarker"
});
```

10.1 status() : String

Return a value hinting at the health of the application:

```
res.send(app.status().concat(" @ ").concat(now());
```

10.2 appctx(path: String) : void

Define the application's `context path` through which requests should be routed. This value is held in the `locals` object.

10.3 assets(path: String) : void

Define the directory, relative to the application's root directory, where `static resources` will be located. This value is held in the `locals` object.

10.4 engine(String view) : void

Define the application's view `rendering engine` - valid values are currently either *freemarker* or *jtwig*. This value is held in the `locals` object.

10.5 resolve(path: String) : String

Sanitizes the path value relative the `appctx` to eliminate ambiguity when dealing with relativity of url paths.:

```
res.redirect(app.resolve("/upload"));
```

10.6 locals() : Set<String>

Return the keys contained in the `locals` object

10.7 locals(param: String) : Object

Return a value from the `locals` object using the given key

10.8 cors(params: Map) : AppServer

Enable and provide request headers to configure *cors* for the application. If an empty or *null* map is provided, the application will use default values to configure the *cors* filter, which are generously open. You can equally turn *cors* on/off through passing a *cors* true/false attribute to the `locals` map.

10.9 router() : AppServer

Create the `AppRoutes` object and return the current instance on `AppServer` to allow method chaining.

10.10 router(supplier: Supplier) : AppServer

This method matches `router()`, except that is allows you to provide your own implementation of the `Router` interface. This allows you to try out different algorithms for matching request handlers to the respective requests.

10.11 filter(filter: HandlerFilter) : AppServer

Apply a filter at the front of the request processing pipeline where the outcome would either allow the request to proceed or send a response instead. This filter is applied to all incoming requests

10.12 filter(context: String, filter: HandlerFilter) : AppServer

Similar to `filter(filter: HandlerFilter)` but this filter is only applied for requests on the specified context

10.13 route(method: String, path: String, handler: HandlerServlet) : AppServer

Add a route to handle requests that match the specified *path* and *method*. This API is targeted for *Java* users.

10.14 route(method: String, path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Similar to the `route(method: String, path: String, handler: HandlerServlet)`. However, this function allows you to dynamically configure the handler before it is added to the server. This is done through the `HandlerConfig` interface. A *Java* example is shown below.:

```
get("/async/{value}", (holder)->holder.getRegistration().setAsyncSupported(true), new␣
↪HandlerServlet() {
    @Override
    public void handle(HandlerRequest request, HandlerResponse response) {
        //code goes here
    }
})
```

In this example, we configure the handler to use servlet 3.0's `AsyncContext` object to process the request by setting `setAsyncSupported` to *true*. This is how we leverage the `AsyncContext` because the default value is *false*.

10.15 head(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle a *head* requests on the *path* url.

10.16 head(path: String, handler: BiFunction) : AppServer

Configures a route to handle a *head* requests on the *path* url. This function is exactly like `head(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.17 head(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *head* requests on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.18 head(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle a *head* requests on the *path* url. This function is exactly like `head(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.19 head(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *head* requests on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `head(path, config, handler: HandlerServlet)` with the added parameters. All the other functions in the `head(...)` family eventually delegate to this function, which add the handler to the server.

10.20 trace(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle a *trace* requests on the *path* url.

10.21 trace(path: String, handler: BiFunction) : AppServer

Configures a route to handle a *trace* requests on the *path* url. This function is exactly like `trace(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.22 trace(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *trace* requests on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.23 trace(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle a *trace* requests on the *path* url. This function is exactly like `trace(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.24 trace(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *trace* requests on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `trace(path, config, handler: HandlerServlet)` with the added parameters. All the other functions in the `trace(...)` family eventually delegate to this function, which add the handler to the server.

10.25 options(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle a *options* requests on the *path* url.

10.26 options(path: String, handler: BiFunction) : AppServer

Configures a route to handle a *options* requests on the *path* url. This function is exactly like `options(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.27 options(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *options* requests on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.28 options(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle a *options* requests on the *path* url. This function is exactly like `options(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.29 options(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *options* requests on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `options(path, config, handler:`

`HandlerServlet)` with the added parameters. All the other functions in the `options(...)` family eventually delegate to this function, which add the handler to the server.

10.30 get(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle a *get* requests on the *path* url.

10.31 get(path: String, handler: BiFunction) : AppServer

Configures a route to handle a *get* requests on the *path* url. This function is exactly like `get(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.32 get(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *get* requests on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.33 get(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle a *get* requests on the *path* url. This function is exactly like `get(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.34 get(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *get* requests on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `get(path, config, handler: HandlerServlet)` with the added parameters. All the other functions in the `get(...)` family eventually delegate to this function, which add the handler to the server.

10.35 post(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle a *post* requests on the *path* url.

10.36 post(path: String, handler: BiFunction) : AppServer

Configures a route to handle a *post* requests on the *path* url. This function is exactly like `post(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.37 post(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *post* requests on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.38 post(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle a *post* requests on the *path* url. This function is exactly like `post(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.39 post(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *post* requests on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `post(path, config, handler: HandlerServlet)` with the added parameters. All the other functions in the `post(...)` family eventually delegate to this function, which add the handler to the server.

10.40 put(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle a *put* requests on the *path* url.

10.41 put(path: String, handler: BiFunction) : AppServer

Configures a route to handle a *put* requests on the *path* url. This function is exactly like `put(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.42 put(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *put* requests on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.43 put(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle a *put* requests on the *path* url. This function is exactly like `put(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.44 put(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *put* requests on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `put(path, config, handler: HandlerServlet)` with the added parameters. All the other functions in the `put(...)` family eventually delegate to this function, which add the handler to the server.

10.45 delete(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle a *delete* requests on the *path* url.

10.46 delete(path: String, handler: BiFunction) : AppServer

Configures a route to handle a *delete* requests on the *path* url. This function is exactly like `delete(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.47 delete(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *delete* requests on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.48 delete(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle a *delete* requests on the *path* url. This function is exactly like `delete(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.49 delete(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle a *delete* requests on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `delete(path, config, handler:`

`HandlerServlet)` with the added parameters. All the other functions in the `delete(...)` family eventually delegate to this function, which add the handler to the server.

10.50 all(path: String, handler: HandlerServlet) : AppServer

Configures a route to handle requests of any *method* on the *path* url.

10.51 all(path: String, handler: BiFunction) : AppServer

Configures a route to handle requests of any *method* on the *path* url. This function is exactly like `all(path, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.52 all(path: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle requests of any *method* on the *path* url. This function also uses the `HandlerConfig` config object to further customize the handler before it is added to the server. With the `config.getRegistration()` object, the handler can be further customized with servlet specific properties to adapt to different requirements.

10.53 all(path: String, config: HandlerConfig, handler: BiFunction) : AppServer

Configure a route to handle requests of any *method* on the *path* url. This function is exactly like `all(path, config, handler: HandlerServlet)` and is more favorable for use with Javascript instead of Java.

10.54 all(path: String, accept: String, type: String, config: HandlerConfig, handler: HandlerServlet) : AppServer

Configure a route to handle requests of any *method* on the *path* url, with a *content-type* of header of *type* value and *accept* header of *accept* value. This function is exactly like `all(path, config, handler: HandlerServlet)` with the added parameters. All the other functions in the `all(...)` family eventually delegate to this function, which add the handler to the server.

10.55 websocket(ctx: String, provider: AppWsProvider) : AppServer

Configure a websocket handler for the application. This will handle websocket requests on the `ctx` context path. The `AppWsProvider` object should provide a prepared instance of `WebSocketAdapter`. You can use `AppWebSocket` which is the the default implementation provided by the framework.

10.56 wordpress(home: String, fcgi_proxy: String) : AppServer

Configure an `FCGI` handler to use with any application that *speaks* fcgi like `Wordpress`.

10.57 listen(port: int, host: String) : void

Fire up the server and start listening for requests on the specified *host* and on the specified *port*.

10.58 listen(port: int, host: String, result: Consumer) : void

This is similar to `listen(port: int, host: String)` except that is also accepts a consumer which gets invoked when the start up is completed.

10.59 lifecycle(event: String, callback: Consumer) : AppServer

Add a lifecycle listener which gets invoked when the corresponding lifecycle even happens. The lifecycle events that will trigger the listener are:

- starting - happens when the server begins to get instantiated
- started - happens when the server is started successfully
- stopping - happens when the server begins to shut down
- stopped - happens when the server has stopped running
- failed - happens when the server is unable to start successfully

CHAPTER 11

App Examples

A few examples are included here to highlight usage scenarios. The syntax used will be JavaScript although the same the Java implementation would almost literary be the same.

11.1 Hello World App

This is just a simple application which responds with the current time when the root context is accessed.

- Create a new project directory and add a `lib` folder to it as well.:

```
mkdir -p my-app/lib;
```

- Download the `zesty-router[version].jar` and place it in the *lib* folder.
- *cd* into *my-app* folder and create a new file, `index.js`.
- In the `index.js` file, import the `AppProvider` class.:

```
let zesty = Java.type('com.practicaldime.router.base.AppProvider');
```

- Initialize the application using some initial configuration.:

```
let app = zesty.provide({});
print('zesty app is configured');
```

- Create a router to add handlers.:

```
let router = app.router();
```

- Add a function to fetch the current time. For a little bit or fun, let's also format the time.:

```
let Date = Java.type('java.util.Date');
let DateFormat = Java.type('java.text.SimpleDateFormat');
```

(continues on next page)

(continued from previous page)

```
function now() {
    return new DateFormat("hh:mm:ss a").format(new Date());
}
```

- Add a route to handle the root context.:

```
router.get('/', function (req, res) {
    res.send(app.status().concat(" @ ").concat(now()));
});
```

- Configure the host and port to listen for requests.:

```
let port = 8080, host = 'localhost';
router.listen(port, host, function(result){
    print(result);
});
```

- Start the application and start serving time.:

```
jjs --language=es6 -ot -scripting -J-Djava.class.path=./lib/zesty-router-0.1.0-
↪shaded.jar index.js
```

11.2 Simple REST App

Let's create a file `simple_rest.js` and begin by creating a simple database access class. Since `nashorn` does not use the `class` keyword, we will use the `function` syntax instead.:

```
let AtomicInteger = Java.type('java.util.concurrent.atomic.AtomicInteger');

function UserDao() {

    this.users = {
        0: {name: "James", email: "james@jjs.io", id: 0},
        1: {name: "Steve", email: "steve@jjs.io", id: 1},
        2: {name: "Carol", email: "carol@jjs.io", id: 2},
        3: {name: "Becky", email: "becky@jjs.io", id: 3}
    }

    this.lastId = new AtomicInteger(3); //this.users.size() - 1

    this.save = (name, email) => {
        let id = this.lastId.incrementAndGet()
        this.users[id] = {name: name, email: email, id: id}
        return this.users[id];
    }

    this.findById = (id) => {
        return this.users[id]
    }

    this.findByEmail = (email) => {
        return Object.values(this.users).find(it => it.email == email )
    }
```

(continues on next page)

(continued from previous page)

```
    this.update = (id, name, email) => {
        this.users[id] = {name: name, email: email, id: id}
    }

    this.delete = (id) => {
        delete this.users[id]
    }
}
```

Next, let's create the API service.:

```
let dao = new UserDao();

let zesty = Java.type('com.practicaldime.router.base.AppProvider');
let app = zesty.provide({
    appctx: '/users'
});

let router = app.router();
router.get('/', function (req, res) {
    res.json(dao.users);
});

router.get('/{id}', function (req, res) {
    let id = req.param('id');
    res.json(dao.findById(parseInt(id)))
});

router.get('/email/{email}', function (req, res) {
    let email = req.param('email');
    res.json(dao.findByEmail(email));
});

router.post('/create', function (req, res) {
    let name = req.param('name');
    let email = req.param('email');
    dao.save(name, email);
    res.status(201);
});

router.put('/update/{id}', function (req, res) {
    let id = req.param('id')
    let name = req.param('name');
    let email = req.param('email');
    dao.update(parseInt(id), name, email);
    res.status(204);
});

router.delete('/delete/{id}', function (req, res) {
    let id = req.param('id')
    dao.delete(parseInt(id))
    res.status(205);
});

let port = 8080, host = 'localhost';
router.listen(port, host, function(result){
    print(result);
```

(continues on next page)

(continued from previous page)

```
});
```

Start the application and listen for requests.:

```
jjs --language=es6 -ot -scripting -J-Dlogback.configurationFile=../lib/app-logback.
↪xml \
-J-Djava.class.path=../lib/zesty-router-0.1.0-shaded.jar simple_rest.js
```

For comparison, the Java equilavent of `simple_rest.js` would be.:

```
public class SimpleRest {

    static class User {

        private int id;
        private String name;
        private String email;

        public User(String name, String email, int id) {
            super();
            this.id = id;
            this.name = name;
            this.email = email;
        }
        //omitted getters and setters
    }

    static class UserDao {

        private AtomicInteger lastId;
        private Map<Integer, User> users = new HashMap<>();

        public UserDao() {
            users.put(0, new User("James", "james@jjs.io", 0));
            users.put(1, new User("Steve", "steve@jjs.io", 1));
            users.put(2, new User("Carol", "carol@jjs.io", 2));
            users.put(3, new User("Becky", "becky@jjs.io", 3));
            lastId = new AtomicInteger(users.size() - 1);
        }

        public Map<Integer, User> all(){
            return this.users;
        }

        public void save(String name, String email) {
            int id = lastId.incrementAndGet();
            users.put(id, new User(name, email, id));
        }

        public User findById(int id) {
            return this.users.get(id);
        }

        public User findByEmail(String email){
            return users.values().stream()
                    .filter(user -> user.getEmail().equals(email))
                    .findFirst()
```

(continues on next page)

(continued from previous page)

```
                    .orElse(null);
        }

        public void update(int id, String name, String email) {
            users.put(id, new User(name, email, id));
        }

        public void delete(int id) {
            users.remove(id);
        }
    }

    public static void main(String...args) {
        UserDao dao = new UserDao();

        Map<String, String> config = new HashMap<>();
        config.put("appctx", "/users");
        AppServer app = AppProvider.provide(config);

        app.router()
            .get("/", (req, res) -> {
                res.json(dao.all());
                return null;
            })
            .get("/{id}", (req, res) -> {
                String id = req.param("id");
                res.json(dao.findById(Integer.valueOf(id)));
                return null;
            })
            .get("/email/{email}", (req, res) -> {
                String email = req.param("email");
                res.json(dao.findByEmail(email));
                return null;
            })
            .post("/create", (req, res) -> {
                String name = req.param("name");
                String email = req.param("email");
                dao.save(name, email);
                res.status(201);
                return null;
            })
            .put("/update/{id}", (req, res) -> {
                String id = req.param("id");
                String name = req.param("name");
                String email = req.param("email");
                dao.update(Integer.valueOf(id), name, email);
                res.status(204);
                return null;
            })
            .delete("/delete/{id}", (req, res) -> {
                String id = req.param("id");
                dao.delete(Integer.valueOf(id));
                res.status(205);
                return null;
            })
            .listen(8080, "localhost", (result) ->{
                System.out.println(result);
```

(continues on next page)

(continued from previous page)

```
            });
    }
}
```

11.3 Adding a Page

Let's now create a home page for the `simple_rest` app we have going. To do this, create a folder `www` in the project's root directory, and add a new file `index.html`.:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Index Page</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <style>
            * {
                margin: 0px;
                padding: 0px;
            }
            #wrapper {
                width: 900px;
                margin: 0px auto;
                display: grid;
                justify-content: center;
                align-content: center;
                grid-template-columns: repeat(3, 20vmin);
                grid-template-rows: repeat(5, 20vmin);
                grid-gap: 10px;
            }
            #wrapper .content {
                display: grid;
                align-content: center;
                justify-content: center;
            }
            #wrapper .content:nth-child(even) {background: #eee}
            #wrapper .content:nth-child(odd) {background: #ccc}
        </style>
    </head>
    <body>
        <div id="wrapper">
            <div class="content">A</div>
            <div class="content">B</div>
            <div class="content">C</div>
            <div class="content">D</div>
            <div class="content">E</div>
            <div class="content">F</div>
        </div>
    </body>
</html>
```

In the `sample_rest.js` file, configure the assets parameters in the `AppProvider`.:

```
let app = zesty.provide({
    appctx: '/users',
    assets: 'www'
});
```

Restart the application and navigate to the root context `http://localhost:8080`. Before adding the `index.html`, the response was a `404 - Not found` error. Now you should expect to see the index page.

11.4 A Freemarker Template

The previous example used a plain `html` page. This example uses a Freemarker template to display the users from the `simple_rest` application. Let's create one. Copy the `index.html` file and rename it to `index.ftl`. This will be layout page for other pages. Let's begin with extracting the css into a new file, `index.css`:

```
* {
    margin: 0px;
    padding: 0px;
}
#wrapper {
    width: 900px;
    margin: 0px auto;
    display: grid;
    justify-content: center;
    align-content: center;
    grid-template-columns: repeat(3, 40vmin);
    grid-template-rows: repeat(5, 40vmin);
    grid-gap: 10px;
}
#wrapper .content {
    display: grid;
    align-content: center;
    justify-content: center;
}
#wrapper .content:nth-child(even) {background: #eee}
#wrapper .content:nth-child(odd) {background: #ccc}
```

Refactor the `index.ftl` page to make it a macro.:

```
<#macro page>
<!DOCTYPE html>
<html>
    <head>
        <title>Index Page</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <link type="text/css" rel="stylesheet" href="/index.css">
    </head>
    <body>
        <div id="wrapper">
            <#nested>
        </div>
    </body>
</html>
</#macro>
```

Now create a template for displaying user data, and call it `users.ftl`.:

```
<#import "index.ftl" as u>
<@u.page>
<#list users?values as user>
<div class="content" data-key="${user.id}">
    <p class="name">${user.name}</p>
    <p class="email">${user.email}</p>
    <p class="link">
        <a href="#" onclick="removeUser(event, '${user.id}')">delete</a>
    </p>
</div>
</#list>
<script>
    function removeUser(e, id){
        e.preventDefault();
        fetch('/users/delete/' + id, {method: 'DELETE'})
            .then(res=> {
                let user = document.querySelector("[data-key='" + id + "']");
                user.remove();
            })
            .catch(err=>console.log(err));
    }
</script>
</@u.page>
```

This template iterates over the values of the users' map passed from the calling function, and for each user it creates a corresponding user element. Each user element also contains a `delete` link. The template contains a script which removes the corresponding user element from the page when clicked. Now create a route to render this `users.ftl` page on the `/users` context.:

```
router.get('/', function (req, res) {
    res.render('users', {users: dao.users});
});
```

And finally, let's configure the `AppProvider` to be aware of the view engine.:

```
let app = zesty.provide({
    appctx: '/users',
    assets: 'www',
    engine: "freemarker"
});
```

Restart the application and navigate to the root context `http://localhost:8080/users`.

11.5 Adding Submit Form

Let's start by splitting the `simple_rest.js` file into two and call the second file `simple_repo.js`. This way, have the repository separate from the rest endpoints, and thereby both file can evolve independently. Let's slightly refector the `simple_repo.js` source code so that it is importable in other modules.:

```
let Dao = {};

;(function(){

    let AtomicInteger = Java.type('java.util.concurrent.atomic.AtomicInteger');

```

(continues on next page)

(continued from previous page)

```
    function UserDao() {

        this.users = {
            0: {name: "James", email: "james@jjs.io", id: 0},
            1: {name: "Steve", email: "steve@jjs.io", id: 1},
            2: {name: "Carol", email: "carol@jjs.io", id: 2},
            3: {name: "Becky", email: "becky@jjs.io", id: 3}
        }

        this.lastId = new AtomicInteger(3); //this.users.size() - 1

        this.save = (name, email) => {
            let id = this.lastId.incrementAndGet()
            this.users[id] = {name: name, email: email, id: id}
            return this.users[id];
        }

        this.findById = (id) => {
            return this.users[id]
        }

        this.findByEmail = (email) => {
            return Object.values(this.users).find(it => it.email == email )
        }

        this.update = (id, name, email) => {
            this.users[id] = {name: name, email: email, id: id}
        }

        this.delete = (id) => {
            delete this.users[id]
        }
    }

    //export the class through the Dao scope
    Dao.UserDao = UserDao;
})();
```

With the split done, now simply import the repository to be used by the endpoints in `simple_rest.js`.:

```
load('./simple_repo.js');

let dao = new Dao.UserDao();

let zesty = Java.type('com.practicaldime.router.base.AppProvider');
let app = zesty.provide({
    appctx: '/users',
    assets: 'www',
    engine: "freemarker"
});

let router = app.router();
router.get('/', function (req, res) {
    res.render('users', {users: dao.users});
});

router.get('/{id}', function (req, res) {
```

(continues on next page)

(continued from previous page)

```
    let id = req.param('id');
    res.json(dao.findById(parseInt(id)))
});

router.get('/email/{email}', function (req, res) {
    let email = req.param('email');
    res.json(dao.findByEmail(email));
});

router.post('/create', function (req, res) {
    let name = req.param('name');
    let email = req.param('email');
    dao.save(name, email);
    res.status(201);
});

router.put('/update/{id}', function (req, res) {
    let id = req.param('id')
    let name = req.param('name');
    let email = req.param('email');
    dao.update(parseInt(id), name, email);
    res.status(204);
});

router.delete('/delete/{id}', function (req, res) {
    let id = req.param('id')
    dao.delete(parseInt(id))
    res.status(205);
});

let port = 8080, host = 'localhost';
router.listen(port, host, function(result){
    print(result);
});
```

With this done, we need to slightly refactor the `users.ftl` file so that we can have a separate template for a single user. Call this new template `user.ftl` and add this markup.:

```
<div class="content" data-key="${user.id}">
    <p class="name">${user.name}</p>
    <p class="email">${user.email}</p>
    <p class="link">
        <a href="#" onclick="removeUser(event, '${user.id}')">delete</a>
    </p>
</div>
```

This template expects a `user` object in its context and renders the user attributes in it. This template will come in handy when creating a new user or even editing an existing user. Now we need to import and use this template in the `users.ftl` file. And while doing so, add another block element for the form to submit user data for persistence in the repository.:

```
<#import "index.ftl" as u>
<@u.page>
<div class="content" data-key="create">
    <form action="/users/create" onsubmit="saveUser(event, this)">
        <input type="hidden" name="id"/>
```

(continues on next page)

(continued from previous page)

```
        <div class="input-row"><span class="title">Name</span><input type="text" name=
↪"name"/></div>
        <div class="input-row"><span class="title">Email</span><input type="text"␣
↪name="email"/></div>
        <div class="input-row"><input class="button" type="submit" value="Save"/></
↪div>
    </form>
</div>
<#list users?values as user>
    <#include "user.ftl"/>
</#list>
<script>
    function removeUser(e, id){
        e.preventDefault();
        fetch('/users/delete/' + id, {method: 'DELETE'})
            .then(res=> {
                let user = document.querySelector("[data-key='" + id + "']");
                user.remove();
            })
            .catch(err=>console.log(err));
    }
    </script>
    </@u.page>
```

The new data entry component we added contains a form which references a `saveUser(event, this)` method. Add the implementation in the `script` section beneath the `removeUser(e, id)` function.:

```
function saveUser(e, form){
        e.preventDefault();
        let id = form.get('id');
        return id? updateUser(id, form) : createUser(form);
}
function createUser(form){
    fetch('/user/create', {method: 'POST', body: form})
        .then(res=>{})
        .catch(err=>{})
}
function updateUser(id, form){
    fetch('/user/update/' + id, {method: 'PUT', body: form})
        .then(res=>{})
        .catch(err=>{})
}
```

We'll add the function bodies in a moment. But before that, add some styling for the form component we just added.:

```
#wrapper .content form {
    padding: 5px;
}
#wrapper .content form .input-row {
    display: flex;
    padding: 5px;
}
#wrapper .content form .title{
    margin: 5px;
}
#wrapper .content form input[type=text]{
```

(continues on next page)

(continued from previous page)

```
    padding: 5px 10px;
    border-radius: 10px;
    line-height: 1.5em;
}
#wrapper .content form input.button{
    padding: 8px 10px;
    min-width: 70px;
    margin: 5px;
}
```

To accomodate these new features, we need to slightly modify the repository in `simple_repo.js`. Let's begin with the `router.get('/{id}'...)` function. Instead of returning a json object, let's have it return a rendered user fragment. This will be useful for both `create` and `update` operations.:

```
router.get('/{id}', function (req, res) {
    let id = req.param('id');
    let user = dao.findById(parseInt(id));
    res.render('user', user);
});
```

Next, modify the `router.post('/create'...)` to redirect after *POST* instead of returning just the status code.:

```
router.post('/create', function (req, res) {
    let name = req.param('name');
    let email = req.param('email');
    let user = dao.save(name, email);
    res.redirect(app.resolve("/" + user.id));
});
```

Next, modify the `router.put('/update/{id}'...)` function to return a rendered component directly. Since the behaviour of a redirect on *PUT* or *DELETE* is not standard across all servers, and because the update does not create a new resource, it makes more sense to respond with the markup instead of redirecting like we did with *POST*.:

```
router.put('/update/{id}', function (req, res) {
    let id = req.param('id')
    let name = req.param('name');
    let email = req.param('email');
    dao.update(parseInt(id), name, email);
    res.render('user', {user: {id, name, email}});
});
```

Now let's add the body for the `createUser(form)` function.:

```
function createUser(form){
    const options = {
        method: 'post',
        headers: {
            'Content-type': 'application/x-www-form-urlencoded; charset=UTF-8'
        },
        body: encodeFormData(form)
    }
    fetch('/users/create', options)
        .then(res=> res.text())
        .then(html=>{
            let parent = document.getElementById("wrapper");
            let element = htmlToElement(html);
```

(continues on next page)

(continued from previous page)

```
            parent.appendChild(element);
        })
        .catch(err=>{
            console.log(err)
        })
}
```

We added an options parameter to describe the request data. We called a new method `encodeFormData()` to convert the form data into a `form-urlencoded` string. Without this step, the data would be sent as `multipart-data` which is not the format we want. In the response, we also called a new method `htmlToElement()` which parses the response into a document element. The implementations for these two helper methods is shown below.:

```
function encodeFormData(data){
    var urlEncodedData = "";
    var urlEncodedDataPairs = [];
    var name;
    for(const name of data.keys()) {
        urlEncodedDataPairs.push(encodeURIComponent(name) + '=' +␣
↪encodeURIComponent(data.get(name)));
    }
    return urlEncodedDataPairs.join('&').replace(/%20/g, '+');
}
function htmlToElement(html) {
    var template = document.createElement('template');
    template.innerHTML = html.trim();
    return template.content.firstChild;
}
```

Now let's add the body for the `updateUser(id, form)` function. Just like we did with the `createUser` method, we define an options parameter to describe the request data, and we use both the `encodeFormData()` and `htmlToElement()` methods in the same way. For the response, this time we replace the existing component with the updated one instead of appending a new one.:

```
function updateUser(id, form){
    const options = {
        method: 'put',
        headers: {
            'Content-type': 'application/x-www-form-urlencoded; charset=UTF-8'
        },
        body: encodeFormData(form)
    }
    fetch('/users/update/' + id, options)
        .then(res=> res.text())
        .then(html=>{
            let parent = document.getElementById("wrapper");
            let element = htmlToElement(html);
            let target = parent.querySelector("[data-key='" + id + "']");
            parent.replaceChild(element, target);
            resetForm();
        })
        .catch(err=>{
            console.log(err)
        })
}
```

For the `updateUser()` to work, we need a way to select a user whom to edit. So we will add a link to the user component that selects the clicked user . In the `user.ftl`, add an edit link like shown below.:

```
<p class="link">
    <a href="#" onclick="selectUser(event, '${user.id}', '${user.name}', '${user.
↪email}')">edit</a>
    <a href="#" onclick="removeUser(event, '${user.id}')">delete</a>
</p>
```

Now we need to add a `selectUser(event, id, name, email)` method in the script section of `users.ftl`. For completeness, let's also add another method, `resetForm()` to clear the form when an update is completed.:

```
function selectUser(e, id, name, email){
    e.preventDefault();
    let form = document.querySelector("[data-key='edit'] form");
    form.elements["id"].value = id;
    form.elements["name"].value = name;
    form.elements["email"].value = email;
}
function resetForm(){
    let form = document.querySelector("[data-key='edit'] form");
    form.elements["id"].value = "";
    form.elements["name"].value = "";
    form.elements["email"].value = "";
}
```

This method populates the *form* component which makes the *Save* operation an update instead of a create operation. At this point, restart the application again and navigate to the `/users` context http://localhost:8080/users.

```
**Please check again soon. The material is continually getting updated**
```

CHAPTER 12

WordPress Example

This example shows how you can configure zesty-router to serve WordPress via FastCGI. The first step is to have WordPress installed on your server machine, for example under /var/www/wordpress. For more information about how to install WordPress, please refer to the WordPress Installation Guide.

This example assumes you are on a Linux system, but there will be a section later on for a Windows system as well. The points below are simply a checklist, and cannot by any measure replace the steps outlined in the wordpress documentation.

- create a holder for the wordpress application `sudo mkdir -p /var/www` with sudo permissions.
- Assuming you have downloaded and unpacked the wordpress application into your home's Download folder, move the wordpress folder to the the *www* directory created in the previous step - `sudo mv ~/Downloads/wordpress /var/www`.
- Install `php-fpm` in your machine using your system's package manager - `sudo pacman -S php-fpm`. This is a *FastCGI Process Manager* for PHP which will bridge the wordpress application to `zesty-router` application.
- Configure `php-fpm` to listen on a TCP port. In some systems, the default configuration is set to listen on a unix socket file which would not work in this case.:

```
# Listen on localhost port 9000
Listen 127.0.0.1:9000
# Ensure only localhost can connect to PHP-FPM
listen.allowed_clients = 127.0.0.1
```

- A quick check using `sudo php-fpm -t` should give an indication as to whether it is configured correctly.
- Start the *php-fpm* process - `sudo systemctl start php-fpm`. You can use *restart* if it was already running.
- You could optionally choose to run *php-fpm* as a service as well if you haven't - `sudo systemctl enable php-fpm`.
- To verify that the process is running at any time, use `sudo ps -ef | grep php-fpm`
- Create a database for the wordpress application, say `db_wordpress`'.

- Configure your database settings in the `wp-config.php` file.

With these steps covered on the wordpress side, let's now configure the `zesty-router` side of the equation. Create your application folder and in it create an `app.js` file. Add this to your new file.:

```
let zesty = Java.type('com.practicaldime.router.base.AppProvider');
let app = zesty.provide({});

let port = 8080, host = 'localhost';
app.router()
    .wordpress("/var/www/wordpress", "http://127.0.0.1:9000")
    .listen(port, host, function(result){
        print(result);
    });
```

We created the app with an empty config to the `AppProvider`, which implies that the root context is */*. We have *php-fpm* listening on port 9000 and wordpress installed at `/var/www/wordpress`. Fire up the app at this point.:

```
jjs --language=es6 -ot -scripting -J-Dlogback.configurationFile=../lib/app-logback.
↪xml \
-J-Djava.class.path=../lib/zesty-router-0.1.0-shaded.jar app.js
```

When the app starts, navigate to http://localhost:8080/wp-admin/install.php to begin setting up wordpress.

CHAPTER 13

WebSocket Example

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C. This example shows how you can configure zesty-router to serve an application using web-sockets.

13.1 Configure A WebSocket Adapter

With plain jetty, there are a variety of ways to create websockets - jetty's own implementation (existed before java had a standard for websockets), javax standards-based implementation and annotation-based implementation, also standards-based. With zesty-router, the implementation used adopted is jetty's own implementation - it's more thoroughly documented and has heavily influenced the javax api for websockets.

To use websockets with zesty-router, it's a simple as creating a class which extends jetty's own WebSocketAdapter, and overriding the desired methods (you get th chose which ones you need, and ignore those you don't need).

Let's create a simple websocket adapter which simply echos back the incoming data, but after transforming it to upper-case.

public class EchoSocket extends WebSocketAdapter { @Override public void onWebSocketClose(int statusCode, String reason) {

System.out.println(getSession().getRemoteAddress().getHostString() + " closed"); super.onWebSocketClose(statusCode, reason);

}

@Override public void onWebSocketConnect(Session sess) {

super.onWebSocketConnect(sess); System.out.println(getSession().getRemoteAddress().getHostString() + "connected");

}

@Override public void onWebSocketError(Throwable cause) {

```
            System.out.println(getSession().getRemoteAddress().getHostString() + " error - " +
            cause.getMessage()); super.onWebSocketError(cause);

        }

        @Override public void onWebSocketText(String message) {

            super.onWebSocketText(message); try {

                System.out.println("Message received - " + message); if(getSession().isOpen()){

                    String response = message.toUpperCase(); getSes-
                    sion().getRemote().sendString(response);

                }

            } catch(Exception e){

                e.printStackTrace(System.err);

            }

        }

    }
```

13.2 Configure a html client (index.html)

Now let's create a html client which connects to this websocket adapter.

```
<!DOCTYPE html> <html lang="en"> <head>

    <meta charset="UTF-8"> <title>Echo Chat</title>

</head> <body> <body>

    <div> <input type="text" id="input" />

    </div> <div>

        <input type="button" id="connectBtn" value="CONNECT" onclick="connect()"
        /> <input type="button" id="sendBtn" value="SEND" onclick="send()" dis-
        abled="true" />

    </div> <div id="output">

        <p>Output</p>

    </div>

</body>

<script type="text/javascript"> var webSocket; var output = document.getElementById("output");
    var connectBtn = document.getElementById("connectBtn"); var sendBtn = docu-
    ment.getElementById("sendBtn");

    function connect() { // open the connection if one does not exist if (webSocket !== undefined

            && webSocket.readyState !== WebSocket.CLOSED) {

        return;

    } // Create a websocket webSocket = new WebSocket("ws://localhost:9001/toUpper");

    webSocket.onopen = function(event) { updateOutput("Connected!"); connectBtn.disabled =
        true; sendBtn.disabled = false;
```

```
            };
            webSocket.onmessage = function(event) { updateOutput(event.data);
            };
            webSocket.onclose = function(event) { updateOutput("Connection Closed"); con-
                nectBtn.disabled = false; sendBtn.disabled = true;
            };
        }
        function send() { var text = document.getElementById("input").value; webSocket.send(text);
        }
        function closeSocket() { webSocket.close();
        }
        function updateOutput(text) { output.innerHTML += "<br/>" + text;
        }
    </script> </body> </html>
```

13.3 Configure A Jetty websocket client

We can also choose to use a java websocket client to test out websocket adapter. To do this, let's create such a client

```
public class ToUpperHandler {
    private Session session;
    CountDownLatch latch = new CountDownLatch(1);
    public void onConnect(Session session){ System.out.println("Connected to server");
        this.session = session; latch.countDown();
    }
    public void sendMessage(String str){
        try{ session.getRemote().sendString(str);
        } catch(IOException e){
            e.printStackTrace(System.err);
        }
    }
    public CountDownLatch getLatch(){ return this.latch;
    }
}
```

Then execute this client using a main method

```
public static void main2(String...args) { String dest = "ws://localhost:8080/upper"; WebSocketClient
    client = new WebSocketClient(); try {
```

```
            ToUpperHandler socket = new ToUpperHandler(); client.start(); URI uri = new URI(dest);
            ClientUpgradeRequest request = new ClientUpgradeRequest(); client.connect(socket,
            uri, request); socket.getLatch().await(); socket.sendMessage("echo");
            socket.sendMessage("test"); Thread.sleep(5000l);

        } catch(Exception e){

            e.printStackTrace(System.err);

        } finally{

            try{ client.stop(); } catch(Exception e){

                e.printStackTrace(System.err);

            }

        }

    }
```

13.4 Configure the application to handle WebSockets

With all those pieces worked out, the only remaining part of the puzzle is wiring these together using zesty-router. This is quite simple and is accomplished by simply adding new websocket routes as shown.

```
    public static void main(String... args) {

        Map<String, String> props = new HashMap<>(); props.put("appctx", "/"); props.put("assets",
        System.getProperty("user.dir") + "/askable-client/www/test");

        AppServer app = AppProvider.provide(props); app.router()

            .websocket("/toUpper", () -> new EchoSocket()) .listen(9001, "localhost", (result) ->
            {

                System.out.println(result);

            });

    }
```

Now if you start the app, you will get to the index.html page, you will get the opportunity to create a websocket connection to the server. Once this is done successfully, you will be able to send messages and he response will be the same message echoed back in upper case.

And that's the gist of it!

CHAPTER 14

License

CHAPTER 15

Need more help?

For more help, reach out directly to zes.ty@aol.com

CHAPTER 16

Indices and tables

- genindex
- modindex
- search