SAVONIA

THESIS - **BACHELOR'S DEGREE PROGRAMME**
TECHNOLOGY, COMMUNICATION AND TRANSPORT

XANDER

A Nim Web Application Development Library and Framework

Author/s: Santeri Sydänmetsä

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study
Technology, Communication and Transport

Degree Programme
Degree Programme in Information Technology

Author(s)
Santeri Sydänmetsä

Title of Thesis

Xander – A Nim Web Application Development Library and Framework

Date	10 August 2019	Pages/Appendices	34

Supervisor(s)
Mr Mikko Pääkkönen, Senior Lecturer
Mr Jussi Koistinen, Senior Lecturer

Client Organisation /Partners
Mr Jussi Nivamo, Research Engineer, Savonia DigiCenter

Abstract

The objective of the thesis was to design and develop a free and open-source web development framework and library for the Nim programming language. As open source projects tend to be large in scale with multiple developers, the aim was to produce a library and framework with stable functionality and a relatively high number of features, with a focus on ease of use.

The project was developed over a period of time by publishing small new features one at a time. Older, working implementations of web libraries and frameworks were studied to understand what features work well and to achieve semantically pleasing and easily understandable code. Taking inspiration from PHP and other languages, Nim's powerful macro system provided the tools to build powerful tools for the developer to use.

The resulting framework reached the set goals of being competent enough for experimental usage. The number of features were adequate and powerful, without compromising ease of use and readability. Future developments were considered for HTTPS and for the make over of the server architecture.

Keywords
Nim, Library, Framework

CONTENTS

ABBREVIATIONS AND DEFINITIONS 5
1 INTRODUCTION 6
2 NIM 7
2.1 History 7
2.2 Features 7
3 THE IDEA BEHIND XANDER 11
3.1 The goals of Xander 11
3.1.1 Inspiration from PHP 11
3.1.2 Inspiration from Go 12
3.1.3 Inspiration from Jester 13
3.2 Ease of use 13
4 DEVELOPMENT WITH THE XANDER LIBRARY 14
4.1 The callback procedure aka. *onRequest* 14
4.2 Preparing request handler parameters 15
4.2.1 Request handler parameter: *data* 15
4.2.2 Request handler parameter: *headers* 15
4.2.3 Request handler parameter: *cookies* 17
4.2.4 Request handler parameter: *session* 17
4.2.5 Request handler parameter: *files* 17
4.3 Compression 18
4.4 Request handlers 18
4.4.1 Creating a request handler 19
4.4.2 Responding 19
4.5 Dynamic routes 20
4.6 Serving files 21
4.7 Routers 22
4.8 Hosts & Subdomains 22
4.9 Helpers 23
4.9.1 *withData* & *withHeaders* macros 23
4.9.2 The *fetch*-procedure 24
4.9.3 requestHook 24

4.9.4 Web sockets 25

4.10 Templates 26

5 WORKFLOW AUTOMATION WITH XANDER 30

5.1 Installing the executable 30

5.2 Running the executable 30

5.3 Creating a project 30

5.4 Running a project 30

5.5 Code listener 31

6 ISSUES DURING DEVELOPMENT 32

7 CONCLUSIONS 33

8 REFERENCES 34

ABBREVIATIONS AND DEFINITIONS

Nim = A statically typed programming language.

Library = An external collection of functions, constants, classes, types etc.

Framework = A set of tools to aid the developer by automating certain tasks.

HTTP = Hyper Text Transport Protocol; data communication for web pages.

Front End = The visible portion of a web application i.e. the user interface, client side.

Back End = The server side of a web application. Not visible to clients.

HTML = Hyper Text Markup Language; a markup language use for web documents.

JavaScript = A high-level, interpreted programming language used to create dynamic features and functionality to websites, such as buttons, pop-ups etc.

PHP = A server-side scripting language used primarily used for web application development

Macro = Macros (or macro systems) allow programmers to access carry out metaprogramming i.e. writing code that produces code. Macros are executed at compile-time.

Procedure = In Nim, a procedure is a function that can alter the state of objects outside of the function's scope.

The programmer = In general, a person using Xander to develop web applications.

1 INTRODUCTION

The goal of this thesis is to create a comprehensive web application development library and framework for the Nim programming language, as none of the kind are easily available as of now. Originally an individual free-time project but due to the scale of the subject, the project will be undertaken for the thesis.

The thesis will be undertaken as an individual project. Due to the nature of the project, Savonia's DigiCenter was assigned to be the "client" of the work.

Xander being a public on-going project, a lot of the features and technologies mentioned in this report are subject to change. Do not use this document as code reference.

2 NIM

Nim is a statically typed general purpose compiled programming language. Syntactically it is very similar to Python, focusing on easy readability with indentation and reserved keywords. Since the Nim compiler firstly compiles the nim code into C, C++, Objective-C or JavaScript, and only then generates an executable, cross-platform functionality is guaranteed. Nim also has a *macro* system, which provides the programmer with a toolkit for manipulating the abstract syntax tree directly. (Rumpf, 2019)

2.1 History

Designed by Andreas Rumph, Nim (formerly Nimrod) appeared for the first time in 2008 as a free and open-source language, making it a relatively young programming language compared to many other commonly used languages, such as Java (1995), PHP (1996) and C# (2000). Although appearing in 2008, Nim only released its stable version as recently as June 2019, having been unstable up till that point with updates that are not necessarily backwards compatible. (Rumpf, 2019)

2.2 Features

Like Python, Nim features indentation, similar constructs and reserved keywords, such as *in*, *is*, *and*, *or*, *where* and *as*. Nim is also garbage collected and provides many garbage collector options, and outperforms many other garbage collector implementations as shown in Figure 1.

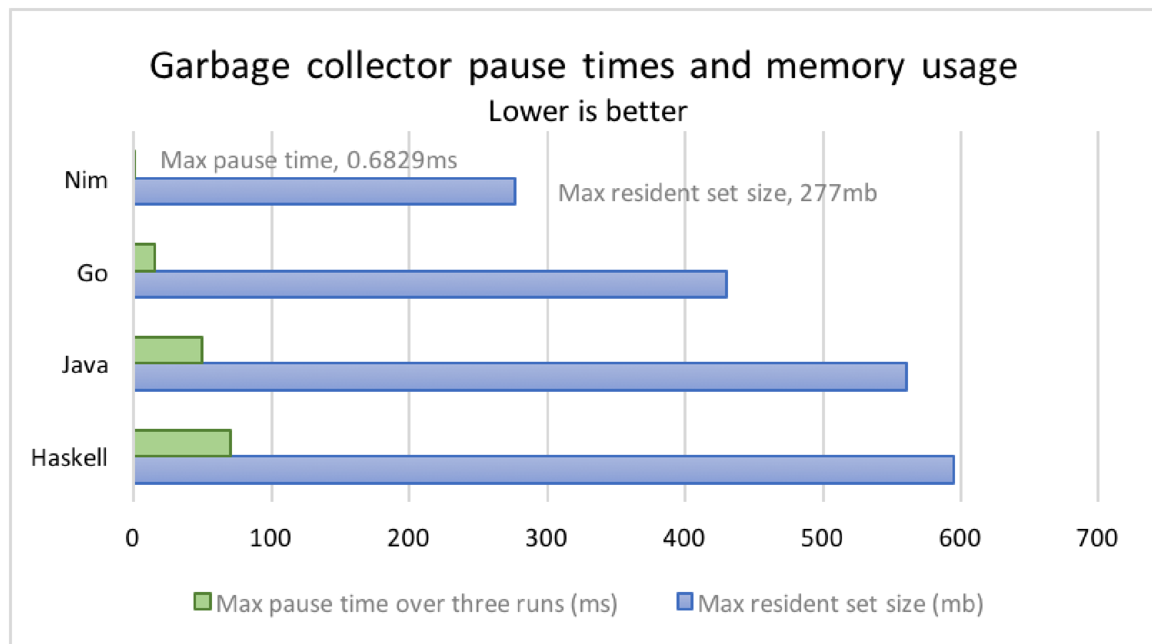

FIGURE 1 Garbage collector performance compared to Go, Java and Haskell. (Rumpf, 2019)

The Nim compiler can compile Nim code to C, C++, Objective-C and JavaScript. This feature allows for several benefits, such as portability and performance (when compiling to C/C++)

as shown in Figure 2 and Figure 3, and the possibility to write front end code in Nim (when compiling to JavaScript in combination with the *dom* Nim module) as shown in Figure 4. Nim also provides the ability to wrap Nim types with external C, C++ and Objective-C types, making it possible to access types and functions from said programming languages.

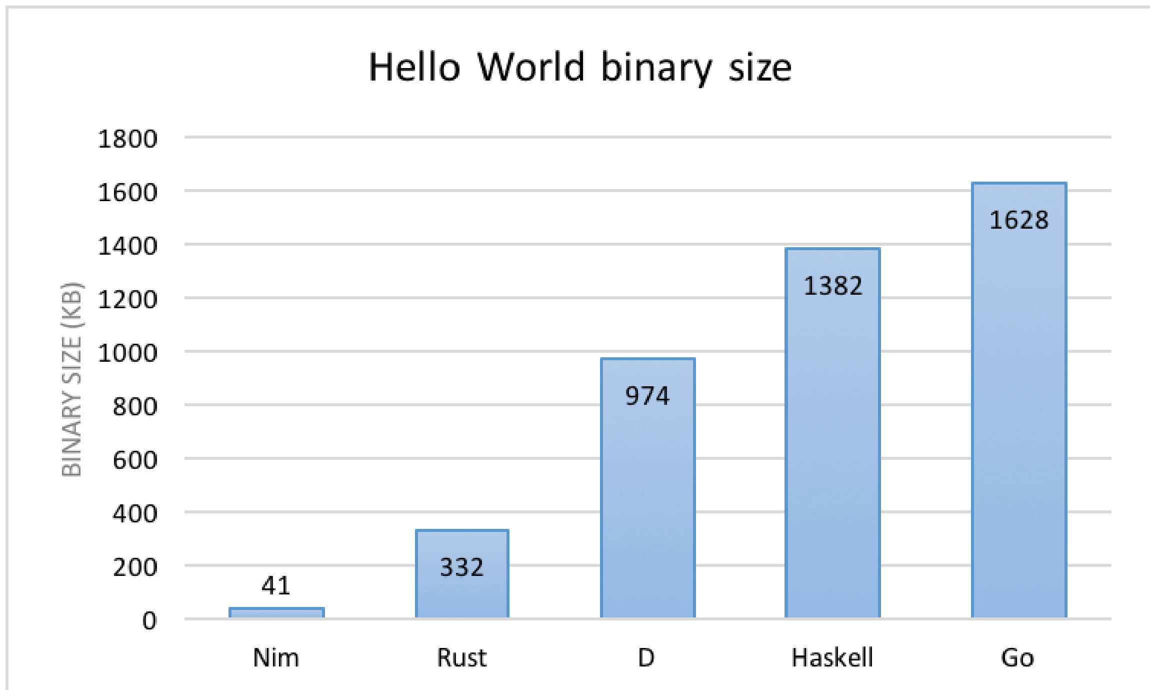

FIGURE 2 "Hello World" application binary size. (Rumpf, 2019)

```
{.passL: "-lsfml-graphics -lsfml-system -lsfml-window".}

type
  VideoMode* {.importcpp: "sf::VideoMode".} = object
  RenderWindowObj {.importcpp: "sf::RenderWindow".} = object
  RenderWindow* = ptr RenderWindowObj
  Color* {.importcpp: "sf::Color".} = object
  Event* {.importcpp: "sf::Event".} = object

{.push cdecl, header: "<SFML/Graphics.hpp>".}
{.push importcpp.}

proc videoMode*(modeWidth, modeHeight: cuint,
                modeBitsPerPixel: cuint = 32): VideoMode
proc newRenderWindow*(mode: VideoMode, title: cstring): RenderWindow
proc pollEvent*(window: RenderWindow, event: var Event): bool
proc newColor*(red, green, blue, alpha: uint8): Color
proc clear*(window: RenderWindow, color: Color)
proc display*(window: RenderWindow)
```

FIGURE 3 Importing C++ types and header file. (Rumpf, 2019)

```
import dom

proc onLoad(event: Event) =
  let p = document.createElement("p")
  p.innerHTML = "Click me!"
  p.style.fontFamily = "Helvetica"
  p.style.color = "red"

  p.addEventListener("click",
    proc (event: Event) =
      window.alert("Hello World!")
  )

  document.body.appendChild(p)

window.onload = onLoad
```

FIGURE 4 Example showing the *dom* module in use. (Rumpf, 2019)

```
proc max(a, b: int): int =
  if a > b:
    return a
  else:
    return b

echo max(3, 2)
echo 3.max(2)
echo 3.max 2

proc print(n: uint) =
  for i in 0 .. n - 1:
    echo "hello!"

3.print
```

FIGURE 5 Examples showing Nim procedure calls. (Rumpf, 2019)

Functions and methods are called “procedures” (or procs) in Nim. Nim supports different kinds of procedure calls: the common way of calling a function by typing the function name with parentheses is naturally supported, as well as procedure calls without parentheses. This makes using some procedures semantically look like accessing properties, as shown in Figure 5.

```
proc max(a, b: int): int =
  if a > b:
    result = a
  result = b


proc max(a, b: int): int =
  if a > b: a else: b
```

FIGURE 6 Returning from procedures in unconventional ways. (Rumpf, 2019)

If a procedure has a return type, Nim will implicitly create a variable in the procedure called *result*, which boasts the same data type as the return type. To make the procedure return a value, simply assigning a value to the result variable is enough, as it does not have to be returned. Also, if the last statement of a procedure has the same data type as the procedure's return type, the last statement will be evaluated and returned as shown in Figure 6.

3 THE IDEA BEHIND XANDER

Nim comes preinstalled with a standard library which provides many powerful and useful procedures and types to develop and build production ready applications. This also applies to web application development, but to effectively develop web applications and get them up and running quickly still requires a lot of set up.

Due to the novelty of the language itself, not many community created web application development libraries exist. The ones that do exist, do not match the specifics that Xander is designed to offer.

3.1 The goals of Xander

As a library, Xander attempts to implement features that other languages or libraries already offer that have been proven to work and improve the programmers work efficiency. Xander aims to allow the programmer to focus on the development of the actual application, instead of using ever so precious time and resources on setting up the backbone of the application. Doing this trades performance for ease of use and an increased speed of development and is an intentional design choice.

3.1.1 Inspiration from PHP

PHP offers implicit global variables (called super globals) for the programmer to use. This means that the programmer never actually initializes said variables with any data themself, and they are always accessible. Knowing this, the programmer can simply access the data they might need with the aid of said global variables. For example, given an HTML form with the following input fields:

```
<form method="POST" action="/">

  <input type="text" name="address">
  <input type="email" name="email">

  <input type="submit" name="addressAndEmail">

</form>
```

FIGURE 7 Form with inputs for *address, email* and *addressAndEmail.*

On the server-side, the data in the input fields can be accessed with:

```
$address = $_POST["address"]
$email = $_POST["email"]
```

FIGURE 8 Accessing form fields in PHP. If the HTML form method is GET, the global variable $_GET would be used.

Certainly, to maintain proper coding standards, the programmer would check that these variables exist before using them. However, the fact that the variables exist implicitly make the developers job easier. The examples in Figure 7 and Figure 8 showed methods of sending simple form data (strings), but several other global variables also exist. These variables include uploaded files, cookie and session variables. (Achour, et al., 2019)

```
$_GET
$_POST
$_FILES
$_COOKIES
$_SESSION
```

FIGURE 9 Useful global variables that exist implicitly in PHP and are designed to be implemented in Xander.

3.1.2 Inspiration from Go

Go (or golang), is a language developed by Google to handle large scale web applications. Setting up a basic server in Go is very straight-forward and can be done with a small set of very readable code. (Golang, 2019)

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", HelloServer)
    http.ListenAndServe(":8080", nil)
}

func HelloServer(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %s!", r.URL.Path[1:])
}
```

FIGURE 10 A basic Go server example. (Golang, 2019)

The developer simply needs to set a "HandleFunc" – a request handler – for a specific URL path and listen to a desired post for requests. This is all achievable with one of the standard Go libraries.

3.1.3 Inspiration from Jester

Jester is currently the go-to web application library for Nim. Developed by one the core developers of Nim, it contains a feature that adds greatly to code readability and simplicity. According to Jester's GitHub page's description, Jester aims to be a "sinatra-like web framework", and certainly, semantically it shows vast similarities to Sinatra, a web library for the Ruby programming language.

```
import htmlgen
import jester

routes:
  get "/":
    resp h1("Hello world")
```

FIGURE 11 Example showcasing a Hello World application made with Jester. (Picheta, 2019)

Syntax-wise the code is very readable: no procedures are defined, and the procedure-calls are short and simple, as shown in Figure 11. As Xander aims to be readable and easy to use, a feature like this is of the highest priority.

3.2 Ease of use

Xander aims to be beginner friendly, all the while providing powerful tools an advanced developer may require. Syntactically, Xander aims to be intuitive and expressive: all procedure and function names are descriptive in what tasks they perform, and are written using whole words, without abbreviations (some exceptions exist due to conflicts with reserved key words in the language itself).

4 DEVELOPMENT WITH THE XANDER LIBRARY

Xander relies on Nim's standard libraries *asynchttpserver* and *asyncdispatch*. These libraries – especially asynchttpserver – are not operating on the lowest level of Nim but provide many useful and powerful tools to build HTTP servers.

```
import asynchttpserver, asyncdispatch

var server = newAsyncHttpServer()

proc cb(req: Request) {.async.} =
  await req.respond(Http200, "Hello World")

waitFor server.serve(Port(8080), cb)
```

FIGURE 12 asynchttpserver Hello World example. Every request responds with a "Hello World" message and HTTP status code 200.

In the Hello World example in Figure 12, an object of the type *AsyncHttpServer* is initialized and bound to the variable *server*. Then, the *serve* method is called with a port number and a callback procedure, which is called every time a request is made to the server. The *respond* procedure in the callback procedure additionally takes an optional argument called *headers*, which contains the response header values.

4.1 The callback procedure aka. *onRequest*

The request callback procedure (called onRequest in Xander) is the core of the event loop, and a wide range of server logic takes place in it. To handle each request, the following tasks are done to prepare the HTTP response:

1. Set default response values (HTTP code, content and headers)
2. Get the response values set by the programmer
3. Merge or replace the two sets of response values

Each HTTP response is made up of three things; HTTP status code, content and headers. These encompass all the data that Xander plays around with. As mentioned in section 3.1.1, Xander is designed to provide the programmer with a set of implicitly defined and initialized variables to do some of the legwork required to run a well-functioning server. Like PHP, Xander defines variables used for handling the request, form data and URL queries, headers, cookies, sessions and uploaded files. However, in Xander these variables are not globals like they are in PHP. Instead, they excist implicitly in every request handler procedure as a parameter (thus the scope is limited to the procedure).

The onRequest procedure calls the programmer-defined request handlers with said variables as parameters, and most importantly, sends them by reference, meaning that all the changes the programmer makes to the variables in the request handler procedure affect the variables in the onRequest procedure as well.

4.2 Preparing request handler parameters

Except for the *request* parameter, which is provided by asynchttpserver, all the other parameters are prepared and assigned some data, if there is any data to be assigned. All the possible incoming data is pulled from the incoming request. The request variable contains a lot of important information about the request. However, the data needed for the Xander defined variables only requires the request's header and body. The request variable is sent to the request handler unchanged as a parameter, as it contains every piece of information regarding the incoming request.

4.2.1 Request handler parameter: *data*

The data parameter is of type *Data*, a data type defined by Xander, and behaves like a JSON object. It is assigned form data and URL query data as key-value pairs. This keeps form and URL query data handling neatly in a single variable. Although also sent in forms, files are handled in a seperate variable.

4.2.2 Request handler parameter: *headers*

The headers parameter is of type *HttpHeaders*, as defined by asynchttpserver. As of currently, the programmer can access the request headers via the request variable, which is sent as a parameter to the request handler. By default, Xander sets some default header fields to ensure that the response bears an adequate level of header information (see Figure 13). Most of the header fields are set to streamline the request-response interaction as well as mitigate attacks targeted towards the server.

```
proc setDefaultHeaders(headers: var HttpHeaders): void =
  headers["Cache-Control"] = "public; max-age=" & $(60*60*24*7) # One week
  headers["Connection"] = "keep-alive"
  headers["Content-Type"] = "text/plain; charset=UTF-8"
  headers["Content-Security-Policy"] = "script-src 'self'"
  headers["Feature-Policy"] = "autoplay 'none'"
  headers["Referrer-Policy"] = "no-referrer"
  headers["Server"] = "xander"
  headers["Vary"] = "User-Agent, Accept-Encoding"
  headers["X-Content-Type-Options"] = "nosniff"
  headers["X-Frame-Options"] = "DENY"
  headers["X-XSS-Protection"] = "1; mode=block"
```

FIGURE 13 The default headers set by Xander

The *Cache-Control* header is set to allow the browser to cache the response, including static files like images and videos as well as text. Setting the maximum age to one week lets the web browser know that the response can be saved as it came for a week and use that save later to speed up later requests. The *Connection* header is set to keep the underlying socket connection alive for further requests. (MDN, 2019)

By default, the *Content-Type* header of the response is set to *text/plain* and character set to *UTF-8*. Thus, the programmer needs but to respond with some text, and it will be displayed correctly, as the character set can display special characters. This header setting has the inevitable effect of failing to display HTML formatted strings correctly in the browser. To work around this, the programmer must use the *html* procedure when responding with a HTML formatted string. (MDN, 2019)

The *Content-Security-Policy* header lets the browser know that any scripts to be run must have their origin on the server, preventing third party scripts from executing in the browser. The *Feature-Policy* header simply prevents the browser from automatically playing any video or audio. The *Referrer-Policy* header omits all referrer information sent with requests. (MDN, 2019)

The *Vary* header requires the caching process to consider the user agent, as well as accepted encoding. This is useful when a web application has versions for desktop and mobile environments. (MDN, 2019)

The *X-Content-Type-Options* specifically helps mitigate XSS attacks by blocking MIME type sniffing. The *X-Frame-Options* header denies iframes from rendering in the browser, preventing clickjacking attacks. The *X-XSS-Protection* header mainly supports the Content-Security-Policy header by mitigating XSS attacks in older browsers. (MDN, 2019)

4.2.3 Request handler parameter: *cookies*

The cookies variable is of type *Cookies*, a data type defined by Xander. The data type consists of two parts: cookies sent by the client and cookies set in the back end. The request cookies are transmitted in the request's header section, and Xander parses these headers to the *cookies* variable. The variable is then accessed as shown in Figure 14.

```
get "/":

  # Retrieves the cookie with the name 'id' from request headers.
  # Request Header: 'Cookie'
  cookies.get("id")

  # Sets a response cookie 'id' with value '123'
  # Response Header: 'Set-Cookie'
  cookies.set("id", "123")
```

FIGURE 14 Getting and setting cookies in request handlers

4.2.4 Request handler parameter: *session*

The session variable is of type *Session*, which is equal with the Xander defined datatype *Data*. The client's session identity (session ID) is stored in the client's browser cookies. The cookie is session cookie is declared *httpOnly*, meaning it cannot be accessed and altered by browser scripts (JavaScript). As of now, the session identity is a SHA-1 hashed random number.

```
cookies.set("XANDER-SSID", ssid, httpOnly = true)
```

FIGURE 15 Xander creating a response cookie declaring the client's session identity

Server sessions are stored in-memory by Xander, and request handlers receive the client's session variables in the *session* parameter.

4.2.5 Request handler parameter: *files*

The files variable is of type *UploadFiles*, a custom data type defined by Xander. Essentially, the data type is a table, where the keys are strings and values are *UploadFile* sequences. The UploadFile data type contains fields for the name, extension, content and size of the file.

```
type
  UploadFiles* =
    Table[string, seq[UploadFile]]
```

FIGURE 16 UploadFiles data type definition

The definition is as shown in Figure 16 to enable forms that send multiple file inputs with different names.

```
<form action="/upload" method="POST" enctype="multipart/form-data">
  <input type="file" name="group_1" multiple/>
  <input type="file" name="group_2" multiple/>
  <input type="submit" value="Upload selected files"/>
</form>
```

FIGURE 17 An example HTML form with two file inputs

```
get "/upload":
  if files.hasKey("group_1") and files.hasKey("group_2"):
    echo "Files in group 1:"
    for file in files["group_1"]:
      echo file.name

    echo "Files in group 2:"
    for file in files["group_2"]:
      echo file.name

    respond "Successfully received both file groups!"
  else:
    respond "One or more file groups missing..."
```

FIGURE 18 File handling in request handlers

4.3 Compression

HTTP compression (decompression in the client) is a common method of improving the speed of interaction between servers and clients. Currently, Xander only supports *gzip* compression (the most common method of compression) and performs the compression process if the *accept-encoding* header of the request contains the value *gzip*. (MDN, 2019)

4.4 Request handlers

Xander defines a custom type called *RequestHandler*, which is a synchronous procedure the onRequest procedure calls on every successful request (the requested URL path exists). The purpose of the request handler procedure is to allow the programmer to define their desired response to be sent back to the client, among other functionality a server may perform.

4.4.1 Creating a request handler

To achieve the simple and readable Sinatra-like look, Xander uses Nim's macro system to essentially trim off all the jargon that goes on behind the scenes. This is necessary, for without the macros, request handler definitions would be too long, both semantically and practically, cluttering the programmers screen with too much unnecessary information.

```
# This...
get "/":
  respond "Hello World!"

# ...turns into this.
addGet("/", proc(r: Request, d: var Data, h: var Headers, c: var Cookies, s: var Session, f: UploadFiles) =
  respond "Hello World!"
)
```

FIGURE 19 Code snippet showing what Xander does to request handler definitions

In Figure 19, the first request handler definition (a macro call) is the preferred and standard Xander way of creating request handlers, whereas the second request handler definition is the actual procedure call to add the request handler to Xander's collection of request handlers. Currently, the transition from *get* to *addGet* (same for other request methods) is carried out by building a string that contains the definition of *addGet*, after which the string is evaluated with a Nim procedure call *parseStmt* (found in the *os* standard library).

Since onRequest – more precicely, the callback procedure parameter of the asynchttpserver library procedure *serve* – must be *gcsafe* (garbage collector safe), variables defined outside of the procedure scope must contain the *threadvar* pragma, if they need to be accessed. This leads to request handlers requiring garbage collector safety, and thus accessing variables outside of said request handler scope requires that the variables are defined with the threadvar pragma.

```
var database {.threadvar.} = newDatabase()

get "/":
  let blogs = database.getBlogs()
  data["blogs"] = blogs
  respond data
```

FIGURE 20 Example showing the threadvar pragma in use

4.4.2 Responding

Technically, the onRequest procedure performs the actual responding, whereas the *respond* procedure called in the request handler simply wraps the programmer defined response into a manageable package for Xander to use as a valid response. The *respond* procedure is overloaded in such a way that the programmer can simply respond with minimal information, such as simply providing

the procedure with an HTTP status code or by providing content in a form of a string, Data or UploadFile. The procedure definitions are as shown in Figure 21.

```
proc respond*(httpCode = Http200, content = "", headers = newHttpHeaders()): Response =
  return (content, httpCode, headers)

proc respond*(content: string, httpCode = Http200, headers = newHttpHeaders()): Response =
  return (content, httpCode, headers)

proc respond*(data: Data, httpCode = Http200, headers = newHttpHeaders()): Response =
  return ($data, httpCode, headers)

proc respond*(file: UploadFile, httpCode = Http200, headers = newHttpHeaders()): Response =
  headers["Content-Type"] = getContentType(file.ext)
  return (file.content, httpCode, headers)
```

FIGURE 21 Request handler response packager procedure definitions

Redirecting has been made possible with the *redirect* procedure as defined by Xander (see Figure 22). The procedure can be used to redirect the user directly to a desired URL, or to redirect after a specified amount of time (in seconds). In HTTP, redirecting is done in one of two ways using headers:

1. Setting the *Location* header to a specified URL.
2. Setting the *Refresh* header to a specified number of seconds to wait and adding the additional parameter *url* and giving it a URL to indicate the new page to redirect to.

```
proc redirect*(path: string, content = "", delay = 0, httpCode = Http303): Response =
  var headers: HttpHeaders
  if delay == 0:
    headers = newHttpHeaders([("Location", path)])
  else:
    headers = newHttpHeaders([("refresh", &"{delay};url=\"{path}\"")])
  return (content, httpCode, headers)
```

FIGURE 22 If the *redirect* procedure is called without providing *delay*, the *Location* header will be used.

4.5 Dynamic routes

Dynamic routes enable the programmer to create endpoints where the specified path is dynamic, opposed to being static. This is achieved in the request handler by adding a colon prefix to the path, and accessing the dynamic value using the *data* variable. This feature is commonly used in addition to other dynamic functionalities a web application may have, such as account creation and access, wikis et al.

```
get "/countries/:country":
  let country = data.get("country")
  if country == "finland":
    respond "Oi Suomi katso, sinun päiväs koittaa..."
  else:
    respond country & " is OK I guess..."
```

FIGURE 23 Using dynamic routes in request handlers

4.6 Serving files

Serving files with Xander is done by calling the *serveFiles* procedure with a specified directory path as its parameter. Any files in possible subdirectories within the specified directory will also be served.

```
serveFiles("/public")
```

FIGURE 24 Example usage of the serveFiles procedure

The way Xander handles file serving is by creating Xander's very own request handlers for the directory and its subdirectories. Using dynamic routes, each request handler parses the requested file's name and checks if the file exists in the specified directory.

```
proc serveFiles*(route: string): void =
  # Given a route, e.g. '/public', this proc
  # adds a get method for provided directory and
  # its child directories. This proc is RECURSIVE.
  logger.log(lvlInfo, "Serving files from ", applicationDirectory & route)
  let path = if route.endsWith("/"): route[0..route.len-2] else: route # /public/ => /public
  let newRoute = path & "/:fileName" # /public/:fileName
  addGet(newRoute, proc(request: Request, data: var Data, headers: var HttpHeaders, cookies:
    let filePath = "." & path / decodeUrl(data.get("fileName")) # ./public/.../fileName
    let ext = splitFile(filePath).ext
    if existsFile(filePath):
      headers["Content-Type"] = getContentType(ext)
      respond readFile(filePath)
    else: respond Http404)
  for directory in walkDirs("." & path & "/*"):
    serveFiles(directory[1..directory.len - 1])
```

FIGURE 25 The serveFiles procedure as defined by Xander

4.7 Routers

Routers help compartmentalize request handlers in a clean and logical order (due to indentation) as well as removing some redundancy of repeating code. A web server may, for example, have its own API endpoints, and thus grouping all the endpoints under one router block makes code more readable and manageable, as shown in Figure 26. The behaviour – produced with macros – is similar to Express' (a Node.js framework) router.

```
get "/":
  respond "Index"

router "/about":
  get "/":
    redirect "/about/us"
  get "/us":
    respond "We are a small scale company."
  get "/sponsors":
    respond "We have multiple sponsors."
```

FIGURE 26 An example router for an *about* section of a web server.

4.8 Hosts & Subdomains

A subdomain is a subsection of a main domain and needs to be configured by the domain name provider. Xander handles subdomains by accessing the request's *host* header value and parsing it for subdomains. Similar to routers, subdomains can help compartmentalize different sections of a server. For example, separating an API from the main domain, as shown in Figure 27.

```
subdomain "api":

  get "/":
    respond "You will find many interesting things in this API..."
```

FIGURE 27 Usage of the subdomain macro. Consider the domain *site.com*, the subdomain will match *api.site.com*

Similar to subdomains, the *host* macro can be used to separate the server into smaller pieces. The macro can be used to serve several different applications with different domain names from the same server. Hosts, subdomains and routers can be layered. The host macro is considered to be experimental.

```
host "localhost":

  get "/":
    respond "Welcome to localhost!"
```

FIGURE 28 Usage of the *host* macro. Requests made to the domain *localhost* will return a welcome message.

4.9 Helpers

Certain macros and procedures were developed to additionally aid the programmer in streamlining their development process.

4.9.1 *withData* & *withHeaders* macros

During the development stage of Xander, some unnecessary redundancies were observed to exist in request handlers. A basic and common function of a server is to receive data from a client and handle it in a certain way. To catch this data on the server side, the developer would have to create a request handler and then explicitly check that the required data was sent, and then possibly assign the data to variables. Due to the frequency of the process mentioned before, a helper macro *withData* was added to streamline the process by implicitly checking if some specified data fields exist and assigning them to variables if they do. A similar helper was created for request headers.

```
post "/submit":

  # Macro for:
  #
  # if data.hasKey("email") and data.hasKey("password"):
  #   var email = data.get("email")
  #   var password = data.get("password")
  #
  withData "email", "password":
    return redirect("/", "Your email is " & email & " and your password is " & password, 5)

  respond Http400
```

FIGURE 29 Usage of *withData*.

4.9.2 The *fetch*-procedure

One function a server may have is retrieving data from a remote location. For example, a weather application may collect weather and climate data from a reputable weather API, or an application may simply have its resources spread into several locations. To access said resources, the programmer would have to use Nim's *httpclient* library to create a request to a remote location. To streamline the process, *fetch*, a PHP-like procedure was created that performs this exact process synchronously in one line.

```
get "/users":
  let users = fetch("https://jsonplaceholder.typicode.com/users")
  respond users
```

FIGURE 30 Usage of the *fetch* procedure

4.9.3 requestHook

As of now, *requstHook* is a variable exported by Xander, which the programmer may use to create a "hook" for Xander's onRequest procedure. The status of requestHook is considered to be experimental. The variable was originally added to allow the programmer to perform asynchronous tasks for client requests. The requestHook can also be used as a section for lower level of web server programming.

```
# EXPERIMENTAL
type RequestHookProc* = proc(r: Request): Future[void] {.gcsafe.}
var requestHook* {.threadvar.} : RequestHookProc # == nil
```

FIGURE 31 requestHook definition. Note, that the variable type is a procedure.

If the requestHook variable is correctly set to be an asynchronous procedure, it (the procedure) will be called whenever a request is made to the server. The requestHook procedure is called first, before any other task is performed by Xander, or by the programmer in a request handler. Technically, the programmer can build their entire application in the requestHook, as it behaves exactly like the callback parameter of the serve procedure in the asynchttpserver library. Calling asynchttpserver's respond method in the requestHook returns from Xander's onRequest procedure, and no other tasks will be performed by Xander.

```
import ../../src/xander

requestHook = proc(r: Request) {.async.} =
  await r.respond( Http200, "Hello World!")

runForever(3000)
```

FIGURE 32 An example application built entirely in the requestHook variable. The application simply returns the message "Hello World!" for every request.

Example use cases for the requestHook procedure are for example, web sockets, asynchronous database tasks and any other asynchronous task. In the development stage of Xander the requestHook was originally used to test web socket functionality. However, web socket functionality was later added to Xander itself.

4.9.4 Web sockets

Web sockets allow the client to communicate with the server – and vice versa - in a more responsive way, where the page does not need to be refreshed after new data is received. A web socket is essentially like an ongoing hands-free phone call; the client can continue to do what ever they might do on the web page, while the web socket connection persists and exchanges data with the server. Web sockets are useful in dynamic and responsive applications, such as live chats, as the client can update the chat's messages everytime the server sends a new message for the client to display, and the client can send new messages to be shown to other recipients. Compared to the likes of AJAX, where the programmer would most likely create a timed call to the server, fetching new data at an interval, the data shown to the client may not be up to date. (MDN, 2019)

In Xander, web socket functionality can be built in the requestHook procedure or the programmer may use the *websocket* macro, which was designed to do some of the legwork for the programmer. The status of the websocket macro is considered to be experimental and uses an experimental community developed web socket library. As of now, the websocket macro only works in the global scope. Thus, a web socket cannot be assigned to a certain host, subdomain or router.

```
websocket "/ws":
  var client = addClient(ws)
  while ws.readyState == Open:
    let packet = await ws.receiveStrPacket()
    let message = "anon#$1: $2".format(client.id, packet)
    await sendMessageToAll(message)
```

FIGURE 33 Usage of the *websocket* macro in the chat server example. Note that the variable *ws* exists implicitly.

In Xander's onRequest procedure, web sockets are second in order of being handled (requestHook being first). The macro implicitly injects a variable called *ws* (short for web socket) into the handler that the programmer may use to access the socket and create their desired functionality.

```
# EXPERIMENTAL
type WebSocketHandler* = proc(ws: WebSocket): Future[void] {.gcsafe.}
var websockets {.threadvar.} : Table[string, WebSocketHandler]
websockets = initTable[string, WebSocketHandler]()

proc handleWebSockets(request: Request): Future[void] {.async,gcsafe.} =
  if websockets.hasKey(request.url.path):
    try:
      var ws = await newWebSocket(request)
      await websockets[request.url.path](ws)
    except:
      discard
```

FIGURE 34 Web Socket definition in Xander

4.10 Templates

Xander houses its own experimental templating engine, which currently provides three main functionalities: importing templates, inserting variables and for loops. The template files themselves can be of any file type, and as Xander's templates are designed to help build dynamic web pages, the .html file type is preferred. The engine uses regular expressions (RegEx) to parse template documents.

By default, Xander looks for templates in a folder called *templates*, which should be located in the application directory. The template directory can be changed by calling *setTemplateDirectory*. In the request handler, the programmer can respond with a template by calling the *tmplt* procedure (this procedure breaks the procedure naming convention, as *template* is a reserved keyword in Nim) with the name of the template. The *tmplt* procedure returns a string containing the page content, and thus this can be returned in the *respond* procedure.

```
# Serve the index page
respond tmplt("index")
```

FIGURE 35 Responding with a template. The *index* template is in the template directory.

A common feature found in many other templating engines is layout files. Layout files declare the layout of a web page, generally including JavaScript and CSS inclusions, meta tags, navigation bars and footers. As these components are static, they should not have to be declared again (to avoid repetition). Thus, creating a single layout file (with the file name *layout*) with the *content tag* is

enough, and Xander will automatically combine a specified template's content with the layout file's content. Xander also supports having multiple layout files, granted that they are separated into separate directories.

```
appDir/
  app.nim
  ...
  templates/

    index.html    # Root page index
    layout.html   # Root page layout

    register/
      index.html  # Register page index
                  # Root page layout

    admin/
      index.html  # Admin page index
      layout.html # Admin page layout

    normie/
      index.html  # Client page index
      layout.html # Client page layout
```

FIGURE 36 An example template directory structure of a Xander application. The directories *register, admin* and *normie* can be considered seperate sections of a server with seperate layout files. The closest layout file will always be served.

Xander defines its template tags as a set of wavy and square brackets with keywords and dynamic values inside. Data passed from request handlers are also accessed in this manner.

```
<!DOCTYPE html>
<html>
  <head>
    <title>{[title]}</title>
  </head>
  <body>
    {[ content ]}
    {[ template footer ]}
  </body>
</html>
```

FIGURE 37 An example layout file. {[title]} is provided by the user. {[content]} will be replaced with template content. {[template footer]} includes a template called *footer*.

Included templates can themselves contain further inlusions of other templates. This makes it possible to build modular and clearly layed out template structures.

```
<!-- templates/footer.html -->
<footer>

  <a href="/">Home</a>

  <!-- templates/contact-details.html -->
  {[ template contact-details ]}

</footer>
```

FIGURE 38 A footer template which further includes a template for contact details.

To pass data to templates, the programmer must call the *tmplt* procedure with an additional parameter, which should be of type Data (JObject). The key-value pairs of the Data object are then injected to the template. Dotwalking is a feature found most notably in JavaScript, and this feature was also implemented in Xander's templating engine.

```
# User requests /countries/ireland/people/paddy
get "/countries/:country/people/:person":
  assert(data["country"] == "ireland")
  assert(data["person"] == "paddy")
  respond tmplt("userPage", data)
)
```

```
<h1>{[person]} is from {[country]}</h1>
```

FIGURE 39 A dynamic end point which returns a template with the dynamic data it received.

For-in-loops can be used to display a list of data, including HTML components. For example, given a list of people, where each person has a *name, age* and *hobbies* field, programmers can generate a table with the list's data by defining a template as follows:

```
<table>
  <tr>
    <th>Name</th>
    <th>Age</th>
    <th>Hobbies</th>
  </tr>

  {[ for person in people ]}
  <tr>
    <td>{[ person.name ]}</td>
    <td>{[ person.age ]}</td>
    <td>
      <ul>
        {[ for hobby in person.hobbies ]}
        <li>{[ hobby ]}</li>
        {[ end ]}
      </ul>
    </td>
  </tr>
  {[ end ]}
```

FIGURE 40 Generating table rows with a for-in loop. Note the JS like dotwalking.

5 WORKFLOW AUTOMATION WITH XANDER

The aim of Xander was to create a web application development library and framework that would help backend developers streamline their development process and additionally get started with a new project quickly without spending an unreasonable amount of time setting up the project environment. For this reason, the main file of the Xander project can also be run as an executable from the command line for some small added benefits.

5.1 Installing the executable

Once the programmer has installed Xander using Nim's own package manager system Nimble, the programmer can optionally (if they wish to use the executable to its fullest potential) install the executable by running a shell script file located in the xander directory. The script creates a hidden directory in the user's home directory and builds the resulting executable there. To run the executable from anywhere, its path must be added to environment variables.

5.2 Running the executable

The executable can be run with three different parameters. The basic format of a Xander executable call is as shown in Figure 41.

```
$ xander [command] [parameters]
```

FIGURE 41 Xander executable running format

5.3 Creating a project

Getting started with a new project can be done by running the xander executable with the *new* parameter. This command creates a new project (called *my-xander-app* by default) by creating an application directory and filling it with useful files to get started with the project, such as the main Nim file and templates. If the programmer wants to name the project, an additional parameter must be provided.

5.4 Running a project

To run a Nim application, the programmer would have to first compile the application, and then run it. This can be done with the nim compiler quite simply, although the resulting command can turn out being quite long.

```
$ nim c -r --hints:off --verbosity:0 app.nim
```

FIGURE 42 Compiling and running using nim c.

For this reason, a simpler way to run the project was developed, so the programmer does not have to memorize and type out the long command. The Xander executable features the *run* command, which by default, if no other parameters are provided, attempts to compile and run the *app.nim* file, which the *new* command creates by default. The command will run the Nim compiler with options as shown in Figure 42.

5.5 Code listener

If any changes are made in the Xander application, the application needs to be compiled and run again to put the changes into effect. To automate this process, the *listen* command was implemented in the executable. The command is run similarly to the *run* command, where the name of the file to run (and in this case, listen for changes) can be omitted. The executable will then proceed to run the application and listen for changes in the applications source code. If any changes are detected, the application will be compiled and run automatically. In case of an error, the error message will be displayed in the command prompt.

6 ISSUES DURING DEVELOPMENT

The development process of Xander as a project contained some issues that were possible to fix or find a work around for. For example, adding the requestHook was necessary to get asynchronous tasks to work, which heavily alters the way developing with Xander works. The work around is more of a hack than a reasonable and well thought out solution. However, it works as it should and does not break the system in any way.

Although HTTPS was one of the planned features of Xander, it was unfortunately not implemented due to issues with asynchronous sockets and a lack of time. In the future, rebuilding Xander from the ground up using lower level libraries, such as starting at the socket level are definite possibilities to ensure HTTPS works natively to Xander.

As of now, a workaround exists to successfully setup HTTPS for Xander applications. The programmer may use other server applications such as Apache or Nginx to setup an SSL proxy that directs secure requests to the Xander application.

7 CONCLUSIONS

The project is still ongoing and will be for some time. However, the current state of Xander is sufficient enough to be used at the very least experimentally but is by no means a complete and stable framework. Xander is an open source project, and thus anyone can aquire the source code of it and contribute as they see fit.

The project was very educational in the rudimentaries of web development and gave me deep insight on how web servers actually work on lower levels of abstraction. I will continue to work on project although development will be less active. The goal is to develop Xander to a steady and stable version where it can be left alone (left for others to further develop).

8 REFERENCES

Achour, Mehdi;ym. 2019. Manual. *PHP.* [Online] 2019. [Cited: 2. August 2019.] https://www.php.net/manual/en/index.php.

Fielding, et al. 1999. RFC 2616. *World Wide Web Consortium.* [Online] 1999. [Cited: 15. July 2019.] https://w3.org/Protocols/rfc2616.

Golang. 2019. Doc. *Golang.* [Online] 2019. [Cited: 22. July 2019.] https://golang.org/doc/.

MDN. 2019. Web HTTP. *Mozilla Developer Network.* [Online] 2019. [Cited: 15. July 2019.] https://developer.mozilla.org/en-US/docs/Web/HTTP.

Picheta, Dominic. 2019. Jester. *GitHub.* [Online] 2019. [Cited: 25. July 2019.] https://github.com/dom96/jester.

Rumpf, Andreas. 2019. Documentation. *Nim-lang.* [Online] 2019. [Cited: 13. July 2019.] https://nim-lang.org/documentation.html.

—. 2019. Features. *Nim-lang.* [Online] 2019. [Cited: 13. July 2019.] nim-lang.org.