Getting Started

This guide assumes that you already have Crystal installed. If not, check out the Crystal installation methods (https://crystal-lang.org/install/) and come back when you're done.

Installing Kemal

First you need to create your application:

```
crystal init app your_app
cd your_app
```

Then add *kemal* to the `shard.yml` file as a dependency.

```
dependencies:
  kemal:
    github: kemalcr/kemal
```

Finally run `shards` to get the dependencies:

```
shards install
```

You should see something like this:

```
$ shards install
Updating https://github.com/kemalcr/kemal.git
Installing kemal (0.21.0)
```

That's it! You're now ready to use Kemal in your application.

Using Kemal

You can do awesome stuff with Kemal. Let's start with a simple example.

```
require "kemal"

get "/" do
  "Hello World!"
end

Kemal.run
```

Running Kemal

Starting your application is easy. Simply run:

```
crystal run src/your_app.cr
```

If everything goes well, you should see a message saying that Kemal is running.

```
[development] Kemal is ready to lead at http://0.0.0.0:3000
2015-12-01 13:47:48 +0200 | 200 | GET / - (666µs)
```

Congratulations on your first Kemal application! This is just the beginning. Keep reading to learn how to do more with Kemal.

Routes

You can handle HTTP methods as easy as writing method names and the route with a code block. Kemal will handle all the hard work.

```
get "/" do
.. show something ..
end

post "/" do
.. create something ..
end

put "/" do
.. replace something ..
end

patch "/" do
.. modify something ..
end

delete "/" do
.. annihilate something ..
end
```

Any **string** returned from a route will output to the browser. Routes are matched in the order they are defined. The first route that matches the request is invoked.

Static Files

Any files you add to the `public` directory will be served automatically by Kemal.

```
app/
  src/
    your_app.cr
  public/
    js/
      jquery.js
      your_app.js
    css/
      your_app.css
    index.html
```

For example, your index.html may look like this:

```
<html>
 <head>
   <script src="/js/jquery.js"></script>
   <script src="/js/your_app.js"></script>
   <link rel="stylesheet" href="/css/your_app.css"/>
 </head>
 <body>
   ...
 </body>
</html>
```

Kemal will serve the files in the `public` directory without having to write routes for them.

Static File Options

Disabling Static Files

By default `Kemal` serves static files from `public` folder. If you don't need static file serving at all(for example an API doesn't need static file serving) you can disable it via

```
serve_static false
```

Static Headers

Adds headers to `Kemal::StaticFileHandler` . This is especially useful for stuff like `CORS` or caching.

```
static_headers do |response, filepath, filestat|
  if filepath =~ /\.html$/
    response.headers.add("Access-Control-Allow-Origin", "*")
  end
  response.headers.add("Content-Size", filestat.size.to_s)
end
```

Modifying Other Options

By default `Kemal` gzips most files, skipping only very small files, or those which don't benefit from gzipping.

If you are running `Kemal` behind a proxy, you may wish to disable this feature. `Kemal` is also able to do basic directory listing. This feature is disabled by default.

Both of these options are available by passing a hash to `serve_static` .

```
serve_static({"gzip" => true, "dir_listing" => false})
```

Views / Templates

You can use ERB-like built-in ECR (http://crystal-lang.org/api/ECR.html) to render dynamic views.

```
get "/:name" do |env|
  name = env.params.url["name"]
  render "src/views/hello.ecr"
end
```

Your `hello.ecr` view should have the same context as the method.

```
Hello <%= name %>
```

Using Layouts

You can use **layouts** in Kemal. You can do this by passing a second argument to the `render` method.

```
get "/:name" do
  render "src/views/subview.ecr", "src/views/layouts/layout.ecr"
end
```

In your layout file, you need to return the output of `subview.ecr` with the `content` variable (like `yield` in Rails).

```
<html>
<head>
  <title>My Kemal Application</title>
</head>
<body>
  <%= content %>
</body>
</html>
```

content_for and yield_content

You can capture blocks inside views to be rendered later during the request with the `content_for` helper. The most common use is to populate different parts of your layout from your view.

Usage

First, call `content_for`, generally from a view, to capture a block of markup with an identifier:

```
# index.ecr
<% content_for "some_key" do %>
  <chunk of="html">...</chunk>
<% end %>
```

Then, call **yield_content** with that identifier, generally from a layout, to render the captured block:

```
# layout.ecr
<%= yield_content "some_key" %>
```

This is useful because some of your views may need specific JavaScript tags or stylesheets and you don't want to use these tags in all of your pages. To solve this problem, you can use `<%= yield_content "scripts_and_styles" %>` in your `layout.ecr`, inside the `<head>` tag, and each view can call `content_for` with the appropriate set of tags that should be added to the layout.

Using Common Paths

Since Crystal does not allow using variables in macro literals, you need to generate another *helper macro* to make the code easier to read and write.

```
macro my_renderer(filename)
  render "my/app/view/base/path/#{ {{filename}} }.ecr", "my/app/view/base/pa
th/layouts/layout.ecr"
end
```

And now you can use your new renderer.

```
get "/:name" do
  my_renderer "subview"
end
```

Filters

Before filters are evaluated before each request within the same context as the routes. They can modify the request and response.

*Important note: This should **not** be used by plugins/addons, instead they should do all their work in their own middleware.*

Available filters:

- before_all, before_get, before_post, before_put, before_patch, before_delete
- after_all, after_get, after_post, after_put, after_patch, after_delete

The `Filter` middleware is lazily added as soon as a call to `after_X` or `before_X` is made. It will **not** even be instantiated unless a call to `after_X` or `before_X` is made.

When using `before_all` and `after_all` keep in mind that they will be evaluated in the following order:

```
before_all -> before_x -> X -> after_x -> after_all
```

Simple before_get example

```
before_get "/foo" do |env|
  puts "Setting response content type"
  env.response.content_type = "application/json"
end

get "/foo" do |env|
  puts env.response.headers["Content-Type"] # => "application/json"
  {"name": "Kemal"}.to_json
end
```

Simple before_all example

```
before_all "/foo" do |env|
  puts "Setting response content type"
  env.response.content_type = "application/json"
end

get "/foo" do |env|
  puts env.response.headers["Content-Type"] # => "application/json"
  {"name": "Kemal"}.to_json
end

put "/foo" do |env|
  puts env.response.headers["Content-Type"] # => "application/json"
  {"name": "Kemal"}.to_json
end

post "/foo" do |env|
  puts env.response.headers["Content-Type"] # => "application/json"
  {"name": "Kemal"}.to_json
end
```

Multiple `before_all`

You can add many blocks to the same verb/path combination by calling it multiple times they will be called **in the same order they were defined**.

```
before_all do |env|
  raise "Unauthorized" unless authorized?(env)
end

before_all do |env|
  env.session = Session.new(env.cookies)
end

get "/foo" do |env|
  "foo"
end
```

Each time `GET /foo` (or any other route since we didn't specify a route for these blocks) is called the first `before_all` will run and then the second will set the session.

Note: `authorized?` and `Session.new` are fictitious calls used to illustrate the example.

Helpers

Browser Redirect

Browser redirects are simple as well. Simply call `env.redirect` in the route's corresponding block.

```
# Redirect browser
get "/logout" do |env|
  # important stuff like clearing session etc.
  env.redirect "/login" # redirect to /login page
end
```

Custom Public Folder

Kemal mounts `./public` root path of the project as the default public asset folder. You can change this by using `public_folder` .

```
public_folder "path/to/your/folder"
```

Logging

Kemal enables logging by default.

You can add logging statements to your code:

```
Log.info { "Log message with or without embedded #{variables}" }
```

You can easily disable logging this like so:

```
logging false
```

Halt

Halt execution with the current context. Returns 200 and an empty response by default.

```
halt env, status_code: 403, response: "Forbidden"
```

Note: `halt` can only be used inside routes.

Custom Errors

You can customize the built-in error pages or even add your own with `error`.

```
error 404 do
  "This is a customized 404 page."
end

error 403 do
  "Access Forbidden!"
end
```

To use a custom `error` you must set the `response.status_code` and then raise a `Kemal::Exceptions::CustomException`.

```
get "/" do |env|
  if some_condition
    env.response.status_code = 403
    raise Kemal::Exceptions::CustomException.new env
  end
  {"message": "Hello Kemal"}.to_json
end

error 403 do
  "Access denied"
end
```

Send File

Send a file with given path and base the MIME type on the file extension or default `application/octet-stream` mime_type.

```
send_file env, "./path/to/file.jpg"
```

Optionally you can override the MIME type

```
send_file env, "./path/to/file.exe", "image/jpeg"
```

For both given examples file will be sent with `image/jpeg` MIME type.

MIME type detection is based on MIME (https://crystal-lang.org/api/0.27.1/MIME.html) registry from Crystal standard library which uses OS-provided MIME database. In case of its absence, it'll use containing basic type list MIME::DEFAULT_TYPES (https://crystal-lang.org/api/0.27.1/MIME.html#DEFAULT_TYPES) as a fallback.

You can always extend registered type list by calling `MIME.register` method with an extension and its desired type.

```
MIME.register ".cr", "text/crystal"
```

Middleware

Middleware, also known as `Handler` s, are the building blocks of `Kemal` . Middleware lets you separate application concerns into different layers.

Each middleware is supposed to have one responsibility. Take a look at `Kemal` 's built-in middleware to see what that means.

Creating your own middleware

You can create your own middleware by inheriting from `Kemal::Handler`

```
class CustomHandler < Kemal::Handler
  def call(context)
    puts "Doing some custom stuff here"
    call_next context
  end
end

add_handler CustomHandler.new
```

Conditional Middleware Execution

Kemal gives you access to two handy filters `only` and `exclude` . These can be used to process

your custom middleware for `only` specific routes, or to `exclude` from specific routes.

```
class OnlyHandler < Kemal::Handler
  # Matches GET /specials and GET /deals
  only ["/specials", "/deals"]

  def call(env)
    # continue on to next handler unless the request matches the only filter
    return call_next(env) unless only_match?(env)
    puts "If the path is /specials or /deals, I will be doing some processin
g here."
  end
end

class PostOnlyHandler < Kemal::Handler
  # Matches POST /blogs
  only ["/blogs"], "POST"

  def call(env)
    # call_next is called for GET /blogs, but not POST /blogs
    return call_next(env) unless only_match?(env)
    puts "If the request is a POST to /blogs, I will do some processing her
e."
  end
end
```

```
class ExcludeHandler < Kemal::Handler
  # Matches GET /
  exclude ["/"]

  def call(env)
    return call_next(env) if exclude_match?(env)
    puts "If the path is not / I will be doing some processing here."
  end
end

class PostExcludeHandler < Kemal::Handler
  # Matches POST /
  exclude ["/"], "POST"

  def call(env)
    return call_next(env) if exclude_match?(env)
    puts "If the request is not a POST to /, I will do some processing her
e."
  end
end
```

Creating a custom Logger middleware

You can easily replace the built-in logger of `Kemal`. There's only one requirement which is that your logger must inherit from `Kemal::BaseLogHandler`.

```
class MyCustomLogger < Kemal::BaseLogHandler
  # This is run for each request. You can access the request/response contex
t with `context`.
  def call(context)
    puts "Custom logger is in action."
    # Be sure to `call_next`.
    call_next context
  end

  def write(message)
  end
end
```

You need to register your custom logger with `logger` config property.

```
require "kemal"

Kemal.config.logger = MyCustomLogger.new
```

That's it!

Kemal Middleware

The Kemal organization has a variety of useful middleware.

- kemal-basic-auth (https://github.com/kemalcr/kemal-basic-auth): Add HTTP Basic Authorization to your Kemal application.
- kemal-csrf (https://github.com/kemalcr/kemal-csrf): Add CSRF protection to your Kemal application.

HTTP Parameters

When passing data through an HTTP request, you will often need to use query parameters, or post parameters depending on which HTTP method you're using.

URL Parameters

Kemal allows you to use variables in your route path as placeholders for passing data. To access URL parameters, you use `env.params.url`.

```
# Matches /hello/kemal
get "/hello/:name" do |env|
  name = env.params.url["name"]
  "Hello back to #{name}"
end

# Matches /users/1
get "/users/:id" do |env|
  id = env.params.url["id"]
  "Found user #{id}"
end

# Matches /dir/and/anything/after
get "/dir/*all" do |env|
  all = env.params.url["all"]
  "Found path #{all}"
end
```

Query Parameters

To access query parameters, you use `env.params.query`.

```
# Matches /resize?width=200&height=200
get "/resize" do |env|
  width = env.params.query["width"]
  height = env.params.query["height"]
end
```

POST / Form Parameters

Kemal has a few options for accessing post parameters. You can easily access JSON payload from the parameters, or through the standard post body.

For JSON parameters, use `env.params.json`. For body parameters, use `env.params.body`.

```
# The request content type needs to be application/json
# The payload
# {"name": "Serdar", "likes": ["Ruby", "Crystal"]}
post "/json_params" do |env|
  name = env.params.json["name"].as(String)
  likes = env.params.json["likes"].as(Array)
  "#{name} likes #{likes.join(",")}"
end

# Using a standard post body
# name=Serdar&likes=Ruby&likes=Crystal
post "/body_params" do |env|
  name = env.params.body["name"].as(String)
  likes = env.params.body["likes"].as(Array)
  "#{name} likes #{likes.join(",")}"
end
```

NOTE: For Array or Hash like parameters, Kemal will group like keys for you. Alternatively, you can use the square bracket notation `likes[]=ruby&likes[]=crystal`. Be sure to access the param name exactly how it was passed. (i.e. `env.params.body["likes[]"]`).

HTTP Request / Response Context

Accessing the HTTP request/response context (query paremeters, body, content_type, headers, status_code) is super easy. You can use the context returned from the block:

```
# Matches /hello/kemal
get "/hello/:name" do |env|
  name = env.params.url["name"]
  "Hello back to #{name}"
end

# Matches /resize?width=200&height=200
get "/resize" do |env|
  width = env.params.query["width"]
  height = env.params.query["height"]
end

# Easily access JSON payload from the parameters.
# The request content type needs to be application/json
# The payload
# {"name": "Serdar", "likes": ["Ruby", "Crystal"]}
post "/json_params" do |env|
  name = env.params.json["name"].as(String)
  likes = env.params.json["likes"].as(Array)
  "#{name} likes #{likes.each.join(',')}"
end

# Set the content as application/json and return JSON
get "/user.json" do |env|
  user = {name: "Kemal", language: "Crystal"}.to_json
  env.response.content_type = "application/json"
  user
end

# Add headers to your response
get "/headers" do |env|
  env.response.headers["Accept-Language"] = "tr"
  env.response.headers["Authorization"] = "Token 12345"
end

# Set response status code
get "/status-code" do |env|
  env.response.status_code = 404
end
```

Context Storage

Contexts are useful for sharing states between filters and middleware. You can use `context` to store some variables and access them later at some point. Each stored value only exist in the lifetime of request / response cycle.

```
before_get "/" do |env|
  env.set "is_kemal_cool", true
end

get "/" do |env|
  is_kemal_cool = env.get "is_kemal_cool"
  "Kemal cool = #{is_kemal_cool}"
end
```

This renders `Kemal cool = true` when a request is made to `/`.

If you prefer a safer version use `env.get?` which won't raise when the key doesn't exist and will return `nil` instead.

```
get "/" do |env|
  non_existent_key = env.get?("non_existent_key") # => nil
end
```

Context storage also supports custom types. You can register and use a custom type as the following:

```
class User
 property name
end

add_context_storage_type(User)

before "/" do |env|
  env.set "user", User.new(name: "dummy-user")
end

get "/" do
  user = env.get "user"
end
```

Be aware that you have to declare the custom type before trying to add with `add_context_storage_type`.

Request Properties

Some common request information is available at `env.request.*`:

- **method** - the HTTP method
    - e.g. `GET`, `POST`, ...
- **headers** - a hash containing relevant request header information
- **body** - the request body
- **version** - the HTTP version

    - e.g. `HTTP/1.1`
- **path** - the uri path
    - e.g. `http://kemalcr.com/docs/context?lang=cr => /docs/context`
- **resource** - the uri path and query parameters
    - e.g. `http://kemalcr.com/docs/context?lang=cr => /docs/context?lang=cr`
- **cookies**
    - e.g. `env.request.cookies["cookie_name"].value`

File Upload

File uploads can be accessed from request but you need to parse it like `HTTP::FormData.parse(env.request) do |upload| end`

It has the following methods

- `name` : The name of the key
- `body` : This is the file for file upload. Useful for saving the upload file.
- `filename` : File name of the file upload. (logo.png, images.zip, etc.)
- `headers` : Headers for the file upload.
- `creation_time` : Creation time of the file upload.
- `modification_time` : Last Modification time of the file upload.
- `read_time` : Read time of the file upload.
- `size` : Size of the file upload.

Here's a fully working sample for reading all the files uploaded and saving it under `public/uploads` .

```
post "/upload" do |env|
  HTTP::FormData.parse(env.request) do |upload|
    filename = upload.filename
  # Be sure to check if file.filename is not empty otherwise it'll raise a c
ompile time error
    if !filename.is_a?(String)
      p "No filename included in upload"
    else
      file_path = ::File.join [Kemal.config.public_folder, "uploads/", filen
ame]
      File.open(file_path, "w") do |f|
        IO.copy(upload.body, f)
      end
      "Upload ok"
    end
  end
end
```

You can test this with below `curl` command.

```
curl -F "image1=@/Users/serdar/Downloads/kemal.png" http://localhost:3000/upload
```

Sessions

Kemal supports Sessions with kemal-session (https://github.com/kemalcr/kemal-session).

```
require "kemal"
require "kemal-session"

get "/set" do |env|
  env.session.int("number", rand(100)) # set the value of "number"
  "Random number set."
end

get "/get" do |env|
  num  = env.session.int("number") # get the value of "number"
  env.session.int?("hello") # get value or nil, like []?
  "Value of random number is #{num}."
end

Kemal.run
```

`kemal-session` has a generic API to multiple storage engines. The default storage engine is `MemoryEngine` which stores the sessions in process memory. You should ***only*** use `MemoryEngine` for development and testing purposes.

See kemal-session (https://github.com/kemalcr/kemal-session) for usage and compatible storage engines.

Accessing the CSRF token

To access the CSRF token of the active session you can do the following in your form:

```
<input type="hidden" name="authenticity_token" value="<%= env.session.string
("csrf") %>">
```

WebSockets

Using *Websockets* with Kemal is super easy!

You can create a `WebSocket` handler which matches the route of `ws://host:port/route`. You can create more than 1 websocket handler with different routes.

```
ws "/" do |socket|

end

ws "/route2" do |socket|

end
```

Let's access the socket and create a simple echo server.

```
# Matches "/"
ws "/" do |socket|
  # Send welcome message to the client
  socket.send "Hello from Kemal!"

  # Handle incoming message and echo back to the client
  socket.on_message do |message|
    socket.send "Echo back from server #{message}"
  end

  # Executes when the client is disconnected. You can do the cleaning up her
e.
  socket.on_close do
    puts "Closing socket"
  end
end
```

`ws` yields a second parameter which lets you access the `HTTP::Server::Context` which lets you use the underlying `request` and `response`.

```
ws "/" do |socket, context|
  headers = context.request.headers

  socket.send headers["Content-Type"]?
end
```

Accessing Dynamic Url Params

```
ws "/:id" do |socket, context|
  id = context.ws_route_lookup.params["id"]
end
```

Testing

You can test your Kemal application using spec-kemal (https://github.com/kemalcr/spec-kemal).

Your Kemal application

```
# src/your-kemal-app.cr

require "kemal"

get "/" do
  "Hello World!"
end

Kemal.run
```

First add `spec-kemal` to your `shard.yml`

```
name: your-kemal-app
version: 0.1.0

dependencies:
  spec-kemal:
    github: kemalcr/spec-kemal
  kemal:
    github: kemalcr/kemal
```

Install dependencies

```
shards install
```

Require it before your files in your `spec/spec_helper.cr`

```
require "spec-kemal"
require "../src/your-kemal-app"
```

Now you can easily test your `Kemal` application in your `spec` s. Create a file called `spec/your-kemal-app_spec.cr` :

```
require "./spec_helper"

describe "Your::Kemal::App" do

  # You can use get,post,put,patch,delete to call the corresponding route.
  it "renders /" do
    get "/"
    response.body.should eq "Hello World!"
  end

end
```

Run the tests:

```
KEMAL_ENV=test crystal spec
```

CLI

A Kemal application accepts a few optional command-line flags:

Short flag	Long flag	Description
`-b HOST`	`--bind HOST`	Host to bind (default: 0.0.0.0)
`-p PORT`	`--port PORT`	Port to listen for connection (default: 3000)
`-s`	`--ssl`	Enables SSL
	`--ssl-key-file FILE`	SSL key file

SSL

Kemal has built-in and easy to use SSL support.

To start your Kemal with SSL support.

```
crystal build --release src/your_app.cr
./your_app --ssl --ssl-key-file your_key_file --ssl-cert-file your_cert_file
```

Deployment

Heroku

You can use heroku-buildpack-crystal (https://github.com/crystal-lang/heroku-buildpack-crystal) to deploy your Kemal application to Heroku.

Capistrano

You can use capistrano-kemal (https://github.com/sdogruyol/capistrano-kemal) to deploy your Kemal application to any server.

Cross-compilation

You can cross-compile a Kemal app by using this guide (http://crystal-lang.org/docs/syntax_and_semantics/cross-compilation.html).

Environment

Kemal respects the `KEMAL_ENV` environment variable and `Kemal.config.env`. It is set to `development` by default.

To change this value to `production`, for example, use:

```
$ export KEMAL_ENV=production
```

If you prefer to do this from within your application, use:

```
Kemal.config.env = "production"
```

When the `KEMAL_ENV` environment variable is not set to `production`, e.g. `development`, an exception page is rendered when an exception is raised which provides a lot of useful information for debugging. However, if the environment variable is set to `production` a standard error page is rendered (see source (https://github.com/kemalcr/kemal/blob/master/src/kemal/helpers/exception_page.cr#L16)).

Note: `KEMAL_ENV` should ***always*** be set to `production` in an production environment for security reasons.

Improve this guide

Please help us improve this guide with pull requests to this website repository (https://github.com/kemalcr/kemalcr.com).

Kemal is developed and maintained by Serdar Doğruyol (https://twitter.com/sdogruyol)

Design by Aşkın Gedik (https://twitter.com/askngdk) and Sinan Mutlu (https://twitter.com/SinanMtl)