Freya is a functional web programming stack for F#. Freya emphasises expressivity, safety, and correctness and is compatible with most server technologies in the .NET space.
CHAPTER 1

Getting Started

The documentation for Freya comes in various forms (as detailed below).

Attention: Getting Started needs completing.
The documentation for Freya is organized into categories to help you find what you’re looking for more easily:

- **Reference** – contains guides to specific libraries and components of Freya, giving a technical view of what is available from each component and how they may be used.

- **Topics** – contains guides specific to certain topics and may cover background information, explanations of design choices and approaches, and may cover multiple parts of Freya when they can be used in concert.

- **Recipes** – contains guides to accomplishing specific tasks with Freya. They may range from the very simple to advanced use-cases, and may span the whole range of the Freya stack.

- **Tutorial** – contains longer form guides, usually covering a complete process of building some web application, and which are likely to cover various elements of Freya along the way.
Contributions to Freya (in any form) are very much welcomed, and if you wish to get involved in any way, see the information on contributing. If you just want to play with Freya and use it for your own things – great! Just remember, if you ever want to get involved, or simply talk about Freya or ask questions – you’ll be welcomed when you do.

3.1 Contributing

Contributions to Freya in any form are very welcome. Whether you just want to correct some grammar in the documentation (!) or you’d like to contribute a new feature, you can feel comfortable that the Freya project will treat you with gratitude and with respect.

**Hint:** If you’re not comfortable with GitHub (or Git in general) or you’re not familiar or comfortable with any of the processes mentioned below, don’t worry. If you contact a maintainer by Email or on Twitter, they’ll help you through anything that you need – everybody needs a helping hand sometimes and nobody knows everything!

3.1.1 Documentation

If you’d like to contribute to the documentation, the quickest way is to use the “Edit on GitHub” link at the top of any page – that’ll allow you to edit any content and send a pull request easily.

Although it’s very likely that you’re help will be accepted without any issue, if you’re planning on making major changes to the structure or content of Freya documentation it’s probably worth discussing it first. Please get in contact via one of the channels below.

3.1.2 Code

As with documentation, the quickest way to contribute is to create a fork on GitHub and submit a pull request. You are also welcome to log issues if you’d like to suggest features, report bugs, or make sure anything else relevant gets noted.

Also as with documentation, if you’re planning on putting significant effort in to a feature or change, please get in contact first, to make sure that it’s likely to be accepted in to Freya – and to make sure it’s not already being worked on!
3.1.3 Contact

Attention: Contact information needs updating.

There are multiple channels you can use to get in touch and talk about Freya with the maintainers and with other users. You should find one to suit you below:

- Gitter

3.2 Help/Issues

3.3 Reference

Reference documentation for Freya contains guides to specific libraries along with guides for using Freya as part of a larger system. These are technical guides and give an overview of general use, rather than a recipe-like approach. The reference documentation is designed to be referred to when you’re using Freya.

The reference documentation is divided into two sections:

- Libraries – contains reference information for the libraries (and sets of libraries) that go to make up Freya itself. This covers the complete set of Freya functionality, and gives a good reference to use when following tutorials and other guidance on Freya.

- Integration – contains information on integrating Freya with the other elements of the web ecosystem such as servers and other web frameworks (one of the features of Freya is that it can integrate seamlessly with most other common components of a .NET web architecture).

When you’re looking for more general information on how to work in Freya, and how to think about solving problems using Freya, you might want to look at some of the other documentation options, particularly:

- Topics – for focused guides to areas of development with Freya.

- Recipes – for Freya-based solutions to common problems.

3.4 Libraries

Library documentation contains reference information for the libraries (and sets of libraries) that go to make up Freya itself. This covers the complete set of Freya functionality, and gives a good reference to use when following tutorials and other guidance on Freya.

3.4.1 Core

Freya.Core provides the basic abstractions on which the Freya stack is built. Most of the types and basic functionality used by the higher level parts of Freya depend on the basics defined in Freya.Core. These essential elements of Freya are defined in the following sections:

Functions

The main abstraction in Freya is an abstraction over the OWIN state – see OWIN. As this is functional programming, you need to pass the state around to functions which require it (and return it from functions which have modified it).
Doing such a common thing manually would become a chore very quickly. The solution is the `Freya<'a>` function, which has the following type:

```fsharp
type Freya<'a> =
    State -> Async<'a * State>
```

In the case of the Hopac variant of the Freya stack – see Hopac – the type is instead defined as:

```fsharp
type Freya<'a> =
    State -> Job<'a * State>
```

The `State` type is a wrapper around the OWIN Environment - it holds a few additional data structures, the relevant part is the OWIN Environment.

A function of type `Freya<'a>` takes the state, and asynchronously (or as a Hopac Job) returns a value of type `'a` and the state. The concurrency options are made available here as one or other of these abstractions is commonly used in web programming. This makes it easy to integrate Freya with existing libraries and software.

**Syntax**

The `Freya<'a>` type would be awkward to implement manually throughout applications, and F# allows for useful syntactic extension in the form of computation expressions.

A `freya` computation expression is provided to make it simpler to write code using Freya, and the functions available throughout Freya are easily usable with this syntax.

**Note:** This syntax is not compulsory – it is possible to use a more operator-based style when writing Freya code if you prefer.

The computation expression syntax looks like this:

```fsharp
let double x =
    freya {
        return x * 2
    }
```

The signature of this function is `int -> Freya<int>`, showing a simple way to work with functions which now have the State threaded through them. Of course, this function doesn’t do anything with the State that it has available – the next section shows how State can be used.

**State**

The important part of the Freya approach to programming is to make programming with state transparent and functional. The most basic way to use information contained within the state is to get the `State` instance and to use some element of it, as seen below:

```fsharp
let readPath =
    freya {
    let! state = Freya.Optic.get id_
    return state.Environment.["owin.RequestPath"] :?> string
    }
```

The first part of this function gets the `State` instance, and the second part extracts some data from it and returns it. Note that the first line uses a `Freya<_>` function – requiring the use of the `let!` syntax for computation expressions.

This very basic usage works, but is not a compelling approach – it can be improved significantly, as detailed in the next section:

- **Optics** – using Optics in Freya for safe, typed data manipulation.
Summary

The basic abstraction of Freya has been introduced, along with the Freya computation expression.

```haskell
// Freya<'a> (State is described)
type Freya<'a> = State -> Async<'a * State> // (or State -> Job<'a * State>)

// Freya computation expression
freya { ... }
```

Optics

In the previous section, you saw that the underlying State instance, and elements of it, could be accessed within a Freya computation expression. While this is workable, it is an untidy and weak approach from the perspective of a strongly typed language.

The solution to this in Freya is to use optics – see the Optics topic for a general introduction – to enable a safer and more functional approach.

Approach

You saw in the previous example (and it’s defined in the OWIN specification) that the data is held within dictionary structures within the state. Here are two ways to retrieve data, the first using the previous naive approach, the second using a pre-defined optic to access the data:

```haskell
// The previous way, using raw access to the state
let readPathRaw =
    freya {
        let! state = Freya.Optic.get id_
        return state.Environment."owin.RequestPath" :?> string }

// The optics way
let readPath =
    freya {
        return! Freya.Optic.get Request.path_
    }
```

The optic approach is clearer and simpler, and also safer – the optic gives type-safe access to the data. Here the Request.path_ optic, defined in the Freya.Optics.Http library, is used to focus on the correct part of the state, and ensure that the data contained is present and correctly mapped to an appropriate type.

The same optic can be used to set the data, or to map a function over the data – the optics are bi-directional. Here’s an example of a function which instead writes to the request path.

```haskell
let writePath path =
    freya {
        do! Freya.Optic.set Request.path_ path }
```

The same lens can be used, the only difference is the function used – Freya.Optic.set versus Freya.Optic.get.

Lenses and Prisms

The OWIN specification makes it clear that not all data in the OWIN Environment will always be present. Some data is optional, so you might find yourself trying to read data that doesn’t exist. In a more general sense, you can construct
optics that may not always be able to traverse a data structure to the data that you want. These lenses are often called prisms, as opposed to the simpler optics – lenses – you saw in the previous section. A lens is an optic to a value of ‘a – a prism to a value of ‘a option.

In versions of Freya prior to 3.0 (due to the way that earlier versions of Aether worked), you needed to use different functions to work with lenses and prisms (then termed lenses and partial lenses). However, from 3.0 onwards, you can use the same methods to work with lenses or prisms, giving a smaller and more consistent API.

Here’s an example using a prism:

```fsharp
let readStatusCode =
  freya {
    return! Freya.Optic.get Response.statusCode_
  }
```

This function is of type Freya<int option>, as the prism returns an option of the value (the response status code is not a required data element in the OWIN specification).

### Morphisms and Types

You might be wondering about the description of optics as providing typed access to the data here. In the underlying data, it is defined in the OWIN specification that this data is stored as an obj (boxed) in a dictionary. How can you work with it as a string, or an int (and in the case of more complex elements of HTTP as a full typed representation of a header, for example)?

The optics defined are composed with morphisms – functions which can convert a data structure to and from another form. In the case above, the response status code is being converted to and from an int transparently as part of the optic access.

This is an important (and powerful) feature of Freya – you can work with strongly typed, expressive representations of data, even though underneath the surface the data is the old string-based web world.

Here’s a quick example, retrieving a header value from the request and receiving a strongly typed representation of that header back, which can be used with all of the type-based F# techniques and tools:

```fsharp
let readAccept =
  freya {
    return! Freya.Optic.get Request.Headers.accept_
  }

// Might return something like...

Some (Accept [
  AcceptableMedia (Open (Parameters (Map.empty)),
    Some (AcceptParameters (Weight 0.3, Extensions (Map.empty))))
  AcceptableMedia (Partial (Type "text", Parameters (Map.empty)),
    Some (AcceptParameters (Weight 0.9, Extensions (Map.empty))))])
```

Here a strongly typed representation of the “Accept” header is retrieved if it’s present – and you’ll receive a fully decomposed, typed representation of that header which you can pattern match, inspect and work with – see Types for more on the type system that Freya uses.

### Summary

The Freya approach to working with stateful data has been defined, giving the common functions for working with data, and some optics that are provided with Freya.
// Get a value from the state using an optic
Freya.Optic.get : optic 'a -> Freya<'a>

// Set a value in the state using an optic
Freya.Optic.set : optic 'a -> 'a -> Freya<unit>

// Map a function over a value in the state using an optic
Freya.Optic.map : optic 'a -> ('a -> 'a) -> Freya<unit>

// Additionally, common Freya provided optics are available in:
open Freya.Optics.Http
open Freya.Optics.Http.Cors

### Pipelines

Functional programming makes much of the ability to compose functions simply – one of the joys of functional programming is being able to compose complex functionality from simpler parts with confidence.

The common Freya<'a> functions are quite simple to use together in various ways (the computation expression syntax being the most obvious – calling another Freya<'a> function within a computation expression is as simple as using the let! or do! syntax. However, it turns out to be useful to introduce another building block, which makes it easier to build systems which compose at a more coarse-grained level.

#### Definition

Freya introduces the concept of a Pipeline function, which is defined like this:

```haskell
type Pipeline = Freya<PipelineChoice>

and PipelineChoice =
  | Next
  | Halt
```

It's simply a normal Freya<'a> function, where 'a is constrained to be PipelineChoice. This is used as a building block throughout various Freya libraries.

#### Composition

The simplest example of this is the basic composition of some Pipeline functions, composed. The Pipeline.compose function can be used to do this. This function has a basic logical model – compositions of pipeline functions will halt when the first value of Halt is returned:

```haskell
let yes =
  freya {
    return Next
  }

let no =
  freya {
    return Halt
  }

// Both of these functions will be run
Pipeline.compose yes no
```
This becomes a useful technique when you want to stop processing a request in a certain situation – for example, you may have written a function which should halt processing if the user making the request is not authorized.

**Operators**

Freya always provides named functions for every piece of functionality, but some parts of libraries lend themselves well to the use of custom operators to make definition more concise and readable. As an alternative to the `Pipeline.compose` function, the infix syntax `>=` is also available. This has the advantage that chains of composition become significantly simpler to read and write:

```plaintext
open Freya.Core.Operators

let functionComposed =
  Pipeline.compose (Pipeline.compose yes no) yes

let operatorComposed =
  yes >= no >= yes

// or if you prefer...

  yes
  >= no
  >= yes
```

It’s a matter of taste and is definitely subjective – but if you find the operator approach clearer, Freya makes them available.

**Summary**

The `Pipeline` function concept has been defined, and a simple approach to composing them.

```plaintext
// Types

type Pipeline =
  Freya<PipelineChoice>

and PipelineChoice =
  | Next
  | Halt

// Composition

Pipeline.compose : Pipeline -> Pipeline -> Pipeline

// Composition (Operator)

open Freya.Core.Operators

(>=) : Pipeline -> Pipeline -> Pipeline
```
Integration

As previously detailed, Freya is built on the OWIN open standard for interoperability between .NET web servers and web frameworks (or stacks).

**Attention:** GitHub issues will need updating.

You should be able to use Freya with any OWIN compatible web server – if you can’t, please raise an issue and we’ll help you however we can.

Functions

For reference for integrating with specific servers, see the Integration documentation. In a general sense however, it is usually only needed to use one function to bridge the Freya and OWIN worlds. We need to be able to turn a Freya function in to an OWIN Application Function, or AppFunc.

```ml
let freyaSystem =
    freya {
    ...
}
let freyaAppFunc =
    OwinAppFunc.ofFreya freyaSystem
```

The OwinAppFunc module contains functions to take a Freya<'a> function and turn it in to an OwinAppFunc. Note that the return value of the function converted will be discarded when it’s run, as there is no use for it in this case (this does mean that any Freya<'a> function can be used).

**Hint:** OWIN specifies signatures for middleware functions – OwinMidFunc – as well as application functions – OwinAppFunc. The OwinMidFunc module contains useful functions, but it is currently considered experimental and is not yet documented.

Summary

The basic conversion of a Freya function to an OWIN compatible function has been demonstrated.

```ml
// Convert any Freya<_> function to an OwinAppFunc
OwinAppFunc.ofFreya : Freya<_> -> OwinAppFunc
```

3.4.2 Types

Freya is driven by a strongly-typed approach to working with the web, and so includes a set of libraries (Freya.Types.*) which model many of the constructs found in web specifications, particularly those dealing with HTTP, URIs, etc. These libraries were previously known as the Arachne project, but were brought under the Freya umbrella in the 3.0 release.

HTTP

The Freya.Types.Http library implements types which represent the semantics of the following standards:

- RFC 7230 – Message Syntax and Routing
This set of RFCs covers basic types present in HTTP requests and responses, principally the data found in the headers of HTTP messages. Strongly typed representations and parsers are given.

To use the types:

```haskell
// Working with the types
open Freya.Types.Http
```

**Note:** Full documentation for the individual type designs within Freya.Types.Http is not currently available, but will be added at a later stage. Inspecting the values returned however should be straightforward and logical, and all typed representations map very closely to the logical design/grammar defined within the appropriate RFC or Recommendation.

**HTTP CORS**

The Freya.Types.Http.Cors library implements types which represent the semantics of the following standards:

- W3C Recommendation on CORS
- RFC 6454 – The Web Origin Concept

The implementation of the Recommendation consists of a set of typed headers. Additionally the Origin header is implemented as defined in RFC 6454. Strongly typed representations and parsers are given.

To use the types:

```haskell
// Working with the types
open Freya.Types.Http.Cors
```

**Note:** Full documentation for the individual type designs within Freya.Types.Http.Cors is not currently available, but will be added at a later stage. Inspecting the values returned however should be straightforward and logical, and all typed representations map very closely to the logical design/grammar defined within the appropriate RFC or Recommendation.

**Language**

The Freya.Types.Language library implements types which represent the semantics of the following standards:

- RFC 4647 – Language Ranges (Matching of Language Tags)
- RFC 5646 – Language Tags (Tags for Identifying Languages)

Strongly typed representations and parsers are given.

No optics are given as a corresponding Freya.Optics.* library as these types are not present directly within HTTP messages, but they are used within some types in the HTTP and HTTP CORS libraries, and may be used directly...
when working with some of the higher levels of abstraction in the Freya stack which expect strongly typed Language Tags/Ranges as configuration values.

```
// Working with the types
open Freya.Types.Language
```

Note: Full documentation for the individual type designs within Freya.Types.Language is not currently available, but will be added at a later stage. Inspecting the values returned however should be straightforward and logical, and all typed representations map very closely to the logical design/grammar defined within the appropriate RFC or Recommendation.

**URI**

The `Freya.Types.Uri` library implements types which represent the semantics of the following standard:

- RFC 3986 – Uniform Resource Identifier (URI): Generic Syntax

Strongly typed representations and parsers are given.

No optics are given as a corresponding `Freya.Optics.*` library as these types are not present directly within HTTP messages, but they are used within some types in the HTTP and HTTP CORS libraries. They are also used in higher levels of abstraction within the Freya stack.

To use the types:

```
// Working with the types
open Arachne.Uri
```

Note: Full documentation for the individual type designs within Freya.Types.Uri is not currently available, but will be added at a later stage. Inspecting the values returned however should be straightforward and logical, and all typed representations map very closely to the logical design/grammar defined within the appropriate RFC or Recommendation.

**URI Template**

The `Freya.Types.Uri.Template` library implements types which represent the semantics of the following standard:

- RFC 6570 – URI Template

Strongly typed representations and parsers are given, along with matching and rendering logic.

No optics are given as a corresponding `Freya.Optics.*` library as these types are not present directly within HTTP messages. These types are used extensively in the Uri Template Routing library, and in work on the representation of Hypermedia standards.

To use the types:

```
// Working with the types
open Freya.Types.Uri.Template
```

Note: Full documentation for the individual type designs within Freya.Types.Uri.Template is not currently available, but will be added at a later stage. Inspecting the values returned however should be straightforward and logical,
and all typed representations map very closely to the logical design/grammar defined within the appropriate RFC or Recommendation.

### 3.4.3 Optics

The `Freya.Optics.*` libraries provide optics from the Freya `State` type to strongly typed properties of the request, response, etc.

**HTTP**

The `Freya.Optics.Http` library provides optics from the `State` to the various aspects of the request and response, modelled using the types from `Freya.Types.Http` (and other Freya.Types.* libraries where needed). These optics are usable directly within a `freya` computation expression, working with the optic functions – see Optics.

The HTTP optics are likely to be the most commonly used optics dealing with request and response data. To use the optic the following modules should be opened:

```haskell
// Working with Freya optics
open Freya.Optics.Http

// Working with the Freya types (maybe required)
open Freya.Types.Http
```

The optics are all provided under the `Request` and `Response` modules (e.g. `Request.path_`), along with sub-modules for headers (e.g. `Request.Headers.accept_`).

**HTTP CORS**

The `Freya.Optics.Http.Cors` library provides optics from the `State` to the various aspects of the request and response modelled using the types from `Freya.Types.Http.Cors` (and other Freya.Types.* libraries where needed). These optics are usable directly within a `freya` computation expression, working with the optic functions detailed in Optics.

These optics are probably not likely to be commonly used, especially when relying on some of the higher level abstractions available in the Freya stack, but they can be useful for writing new low-level code.

```haskell
// Working with Freya optics
open Freya.Optics.Http.Cors

// Working directly with the types if required
open Freya.Types.Http.Cors
```

The optics are all provided under the `Request.Headers` and `Response.Headers` modules (e.g. `Request.Headers.acceptControlAllowOrigin_`).

### 3.4.4 Routers

The `Core` library, along with the `Types` and `Optics` libraries, allow for the creation of powerful handlers for HTTP requests, but they don’t provide support for routing requests to specific handlers.

Routers are used to solve this problem, and allow for the aggregation of multiple handlers in to a more complex application. Freya allows for the possibility of multiple different implementations of routing, to support differing requirements. Currently Freya includes one router, based on URI Templates.
• **URI Template** – a router which efficiently uses URI Templates to dispatch HTTP requests to handlers, and exposing the matched data as strongly typed URI Template data types.

**URI Template**

**Defining Routes**

The Freya router is based on mapping requests to FreyaPipeline functions (see /core/pipelines if you’re not familiar with the Freya pipeline concept). It uses a custom computation expression (simpler and more limited than the common freya computation expression) tailored specifically for defining routing – the freyaRouter computation expression.

**Syntax** The computation expression uses a custom operation to let you define routes in a simple, typed way (the router is also extended by other Freya stack libraries, as you will see in other sections, but we will concentrate on the simple, common case here).

By default, we define routes by specifying a requirement for the HTTP method (or verb), a requirement for the path, and the FreyaPipeline function to call if the route is matched. Route matching effectively happens in definition order precedence, although the router internally converts the route definitions to an optimised trie, so this is not literally true!

We use strongly typed values for the requirements, using types taken from the /types-and-optics/arachne libraries, in this case Arachne.Http and Arachne.Uri.Template, as well as types from Freya.Router. Here’s an annotated example of setting up a router, including opening appropriate modules:

```ocaml
open Arachne.Http
open Arachne.Uri.Template
open Freya.Core
open Freya.Router

// Handlers

let handlerA =
  freya { return Next }

let handlerB =
  freya { return Next }

// Routing

let routeA =
  UriTemplate.parse "/a/{id}"

let routeB =
  UriTemplate.parse "/b/{name}"

let routes =
  freyaRouter {
    route Any routeA handlerA
    route (Methods [ GET; OPTIONS ]) routeB handlerB
  }
```

There’s quite a lot going on here, let’s look at the parts in turn to break down what’s going on. First of all, our two handlers are simply FreyaPipeline functions:
Next we come to the first critical aspect of routing – the use of URI Templates. URI Templates are generally used to turn a template and some content into a URI (or some part of a URI, as the specification does not actually specify that a rendered template must be a well-formed URI).

However, they can also be used for matching (with some caveats) and in Freya we use URI Templates for both matching and rendering (this will be useful in higher levels of abstraction, such as Hypermedia generation).

Here we define two URI Templates, which will match the paths shown. The syntax for a match is quite obvious here – we’re doing a simple match, capturing the braced terms (e.g. `{id}`).

```plaintext
let routeA : UriTemplate = 
    UriTemplate.parse "/a/{id}"

let routeB : UriTemplate = 
    UriTemplate.parse "/b/{name}"
```

Finally, we define the routes themselves, using the `route` keyword in the computation expression. This takes three arguments:

- A `FreyaRouteMethod`, which may be `All` – matching any method, or `Methods` which takes a list of Method values which are allowed. In the example, the first route will match any method, the second only GET or OPTIONS requests.
- A `UriTemplate` which will be matched against the request path.
- A `FreyaPipeline` which will be called if the method and the path are matched.

```plaintext
let routes = 
    freyaRouter { 
        route Any routeA handlerA 
        route (Methods [ GET; OPTIONS ]) routeB handlerB 
    }
```

The router will call the `FreyaPipeline` function of the first matched route. If no route matches, no pipeline will be called.

**Type Inference** Freya 3.0 introduced a more extensive use of static type inference to give more concise and flexible APIs. One of the places where this is used is in the `freyaRouter` computation expression. Rather than only taking a literal `FreyaRouteMethod` as the first parameter, the `route` function can actually take any value which has a static `FreyaRouteMethod` member. By default, this is defined for a few different types, all of which can be used interchangeably. That means that the following are all valid routes:

```plaintext
let routes = 
    freyaRouter { 
        route Any routeA handlerA // Any 
        route (Methods [ GET; POST ]) routeA handlerA // Methods 
        route [ GET; POST ] routeA handlerA // Method list, inferred 
        route GET routeA handlerA } // Method, inferred 
```

As we can see, this can make things more concise and more readable! In addition to the Methods, both the template and the handler are also inferred - anything which has `UriTemplate` and anything which has `FreyaPipeline` can be used. By default, this means that you can just use strings and they will be statically inferred as templates, and
any Freya function can be inferred as a Pipeline. This can mean that a previously more complex configuration can be made much simpler:

```fsharp
let routes1 =
    freyaRouter {
        route (Methods [ GET ]) (UriTemplate.parse "/hello") handlerA }

// is the same as...

let routes2 =
    freyaRouter {
        route GET "/hello" handlerA }
```

**Pipeline** In earlier versions of Freya, it was neccessary to call an explicit `toPipeline` function to use a Router as a pipeline. This is no longer needed in 3.0+. (FreyaRouter implements `FreyaPipeline` and thus anything which expects to be able to infer a pipeline can accept a FreyaRouter).

**URI Templates** In our router example here, we have defined our URI Templates separately from the router, and referred to them. We could of course do this inline, saving space. However, it is often useful for multiple parts of a program to be able to refer to the URI Template as a first class item, so we commonly define them outside of the router itself.

This becomes especially useful when you wish to return the URI of a resource as part of a response. You can use the same URI Template for routing and generating linking URIs, which prevents the two ever becoming unsynchronised, using the typed approach to prevent a class of error.

Of course, to do so we will need to know how to work with URI Template data, both for rendering, and for interpreting routing matches in our handlers, which we will see in the next section.

### 3.5 Integration

Integrating Freya with other systems (whether servers to host Freya applications, or other software frameworks) is essential. Freya should integrate widely through the use of the OWIN open standard, but it is not always obvious how it should work in practice. Examples and information for integration are given in the two sections of this reference:

- **Frameworks** – integrating Freya with other frameworks, for example the Suave web framework. Freya plays nicely with others, and further and better integration is always a goal.
- **Servers** – integrating Freya with servers for hosting. While OWIN is an open standard, the approaches to integrating OWIN vary between server implementations.

#### 3.5.1 Frameworks

Currently documented integrations:

- **Suave**

**Note:** Documentation on Suave integration is coming soon.
3.5.2 Servers

Currently documented integrations:

Katana

Note: Pull requests to Freya documentation adding or extending information on usage with any particular server are particularly welcome. Freya should be broadly compatible, and every effort will be made to solve compatibility issues if they are discovered.

It is very simple to integrate Freya applications with the Katana server, especially using the SelfHost options. A simple example is show below:

```fsharp
// Freya
open Freya.Core
let application =
    freya {
        ...
    }
let owinApplication =
    OwinAppFunc.ofFreya application
// Katana
open Microsoft.Hosting

type Application () =
    member __.Configuration () =
        owinApplication
// Main
[<EntryPoint>]
let main _ =
    let _ = WebApp.Start<Application> ("http://localhost:8080")
    let _ = System.Console.ReadLine ()
```

This will give a Katana based server running Freya-delivered content in a console application.

Kestrel

3.6 Topics

3.7 General

3.7.1 Hopac

The “default” implementation of Freya uses F# async functions for most operations, and the internals are built as async. However, there is an alternative implementation available using the Hopac concurrent programming library
available.

The Hopac version of Freya is effectively identical in functionality and usage, but uses a different underlying concurrency model – see Functions. This can make Freya easier to integrate with existing code where Hopac is in use, and also potentially gives some performance/resource usage gains (the Freya Benchmarks project is beginning to measure these reliably).

### Packages

Packages which use the Hopac concurrency model are suffixed with .Hopac. If you are using the Freya meta-package for example, rather than taking a dependency on Freya, you would take a dependency on Freya.Hopac. This convention applies to all packages available in both variants.

**Note:** Not all packages are dependent on a concurrency abstraction, so not all packages come in both “default” and Hopac variants. Packages which do cannot be mixed and matched – attempting to use one package built for async and one built for Hopac together will result in errors!

### Reference

For reference and more information on Hopac:

- Hopac – the Hopac concurrency library for F#, based on ConcurrentML.
- Hopac Documentation – reference documentation on Hopac.

#### 3.7.2 Optics

Optics (often – and in earlier versions of Freya – referred to as lenses) are a functional technique to enable you to work with complex data structures more easily. As data structures are generally immutable, modifying a data structure (or returning a new instance with the changes reflected) can be quite onerous if the data structure is large and complex, and the area you wish to change is deep within the data structure.

Optics, as their name implies, enable you to focus on a particular part of a data structure, treating it as if it were a normal top level instance of some data. You can think of doing your work through a lens (an optic), which will handle the mechanics (or optics, rather) of data structure modification for you.

#### Aether

Freya uses the Aether optics library. The guides to optics found there give a good introduction to the principles, along with more general information about the library itself (which is used extensively in Freya).

- Aether Guides – guides to the principles and usage of optics, with examples and explanation.
- Aether – the main Aether website.

#### Freya

Optics are a key part of Freya usage, used to work with data throughout many elements of the Freya stack. They are discussed in the reference documentation, as well as being used throughout any tutorial or recipe documentation.

- Optics – reference for optic usage in the Freya.Core library.
3.7.3 Upgrading

3.8 Standards

3.8.1 OWIN

3.9 Recipes

3.10 Tutorial