Freya Documentation

Release 3.0

Andrew Cherry, Ryan Riley

September 05, 2016

Topics

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 quickest way to get up and running with Freya and to get a feeling for what programming with Freya is like, is to dive straight in to the Getting Started tutorial. From zero to a Freya-based Hello World in a few minutes! Once you're ready to move on, you can move on to the more in-depth documentation, as described below...

Hint: The Getting Started tutorial is the perfect way to take your first steps with Freya.

CHAPTER 2

Sections

The documentation for Freya is organized in to sections to help you find what you're looking for more easily:

- 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.
- Tutorials – 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.
- 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.
- 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.

2.1 Meta

For more "meta" topics around the Freya project and community (such as contact information, guidance on contributing, etc.) see the following sections:

2.1.1 Community

The F# community is a friendly place and we hope for the same for Freya. We encourage everyone to come and talk in our Gitter channel where all are welcome. It's a great place to ask questions!

For keeping up to date with news on Freya and the ecosystem around it, follow Freya on Twitter – any news, changes, announcements, releases, etc. will be posted there, along with new blog posts, etc.

Most of all, we encourage you all to get involved, try Freya out, and tell us what you think!

2.1.2 Contact

There are several ways to get in touch!

Help/Advice

Probably the quickest way to ask a question and chat it through is to ask on the Freya Gitter channel – while it's not always going to be answered immediately, people are often available, and responses don't usually take too long! If you're on the FSSF Slack channels, some of the maintainers are also often available there.

Issues/Features

If you want to raise an issue relating to Freya, or suggest a new feature – or an improvement to an existing one – then the main Freya repository on GitHub should be your first stop.

Issues, feature requests, etc. for Freya are tracked and managed using the GitHub Issues process. While Freya is split in to multiple repositories, issues raised against the main repository can be moved as needed, and it is often easier to work out where they should live later!

Miscellaneous

If you just want to send an email to a human being, then that's possible too. Email freya@xyncro.com for an open-ended channel.

2.1.3 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're involvement will be appreciated and respected. If you're not sure how you'd like to be involved, but feel that you would in some way – get in Contact. Regardless of experience, skill level or time available, there's probably something for you!

Hint: If you're not comfortable with GitHub – or Git in general – or you're not familiar 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!

Documentation

If you'd like to contribute to the documentation, the quickest way is to use the "Edit Page" link at the bottom of any page – that'll allow you to edit any content and send a pull request easily via GitHub.

Although it's very likely that your 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 and talk before expending major effort!

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.

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!

2.1.4 Policies

Freya does not have a large set of policies, but some aspects of the project do require explicit affirmation of expectations. Please read the following on conduct!

Conduct

While Freya does not currently have an explicit code of conduct, any behaviour which prevents others from finding Freya and the Freya community a welcoming and friendly environment will not be tolerated. If you are unsure whether something is likely to be offensive or upsetting to others, that is usually a good indication that you probably shouldn't do it.

Were Freya to adopt a formal code of conduct, it would probably look rather like the Rust Code of Conduct – please refer to that document for the kinds of behaviour expected of people taking part in the Freya community.

Open source software matters, but it matters less than people.

2.2 Topics

Topic guides are available covering various aspects of building with Freya. These guides deal broadly with usage, background information, history, design decisions, etc. The topic categories are available directly from the top level navigation.

2.3 Design

The design of some aspects of Freya merits particular discussion – understanding the approaches taken can help get the most out of the available components of the Freya stack.

In particular, the following sections are key to some elements of the Freya stack:

- Machines – the design of Freya Machines merits in-depth explanation – it is a powerful and extensible approach to web programming.

2.3.1 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.

2.3.2 Machines

Freya Machines are a powerful tool, but the programming model can be new to many, especially given the prevalence of web frameworks (particularly MVC-style frameworks) where the logic is spread throughout the whole framework. Machines can therefore look daunting at first, but they actually represent a conceptually simpler and more straightforward approach.

The following references give a directed explanation of Machine design and implementation in Freya, using an HTTP Machine as a convenient example where relevant.

- Decisions – decisions are the basis of the Machine programming model. Structuring logic as well-defined decisions gives more concision and clarity while (when combined with a suitable type model) also giving opportunities for powerful optimisations.
- Configuration – configuration of a Machine is the means by which it can be accurately tailored to a specific purpose. Configuration can be simple and static or dynamic – enabling a surprisingly powerful execution model.
- Optimisation – a structured decision model combined with suitable type information enables a useful and well-defined optimisation approach to the eventual logic. Machines can be optimised to only execute the minimum neccesary execution path, giving efficient implementations of complex logic.

Decisions

Note: This documentation is currently being written and reviewed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

Configuration

Note: This documentation is currently being written and reviewed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

Optimisation

Note: This documentation is currently being written and reviewed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

2.3.3 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.

2.4 Standards

Freya is an intentionally standards based stack, whether that is broad public standards (like RFCs for HTTP, etc.) or community based, like OWIN. More on standards and how Freya works with them can be found in the following sections:

2.4.1 OWIN

OWIN (Open Web Interface for .NET - see owin.org) is a simple, community driven, standard interface between web servers and web applications. With a simple abstraction in place, people can concentrate on writing servers or applications, confident that they will run happily with a server/application that also supports the OWIN standard.

Interface

The OWIN standard interface is not a particularly complex one, as it's designed to be a fairly "lowest common denominator" approach, with the aim of being applicable to the broadest possible ecosystem (so no relying on .NET features only accessible from specific languages, minority versions or frameworks, etc.)

The standard, in basic form, looks like this (straying in to C# for a moment):

```
using AppFunc = Func<IDictionary<string, object>, Task>;
```

In effect it's simply saying "I'm going to give you a dictionary containing request and response data. You muck about with that, and return me a Task when you're done with it." The server is then responsible for taking the eventual dictionary (called the *Environment*) and making sure that it gets turned in to a valid HTTP response, and sent back to the client.

Data

The obvious question now is "Where's the request and response data?" Well, it's in the *Environment*, boxed up with string keys. Some of the values are boxed strings, some boxed dictionaries, some boxed streams... It isn't elegant, but it is workable, and making it this low level is one of the only ways to give an interface that multiple languages can actually work with.

As an example, the key **owin.RequestPath** in the *Environment* will get (or set, as it's just a dumb dictionary) the path of the request. The key **owin.ResponseBody** is the stream used to write the response body, and so on.

As you'll note – if you've got an F# hat on (our merchandise store is coming, we assure you) – this is all based on mutation of the *Environment*. That's not a lovely thing to see in F#, and Freya does quite a lot of work to tidy this up (or at least hide it where you can pretend it doesn't happen). You can see more about how it does so in the Core Reference section – see Core.

Servers

As you can see from the simplicity of the interface (known as the *AppFunc* or *OwinAppFunc*), there's not much that a library needs to be able to do to work with a compatible OWIN server. Different OWIN servers, however, have different ways of asking for that *AppFunc*, so you may need to peruse the documentation for your specific server.

In the examples and tutorials we'll see later, we'll often use the Katana server, and we'll show how simple it is to host Freya using that. If you're using something else, it may be documented – see Servers. If not, feel free to either ask for guidance or even submit documentation – see Contributing!

2.4.2 Polyfills

The OWIN standard is a good base point for developing interoperable servers and frameworks, but like any standard it requires iteration and development. Sometimes it lacks features which Freya requires to work as well as possible – and in those cases Freya provides polyfills for particular server implementations which will "extend" the OWIN standard with additional features which Freya code can then test for and use where present.

This enables Freya to drive forward and improve at a fast pace.

For documentation on the polyfills currently available for Freya see the library reference.

- Polyfills – available polyfills for Freya, providing extensions to standards such as OWIN.

2.5 Versioning

Freya follows a semantic versioning approach to versioning, with some key points, due to the multi-library nature of the Freya stack.

The overall version of Freya (and the version used to define versions of documentation, etc.) is the version of the Freya meta-package – the package which has dependencies that make up a default Freya stack. The version of this meta-package may be increased driven by changes to the dependency set.

In effect, this means that a breaking change to a dependency will likely require a major version increase of that dependency. The next release of the Freya meta-package which includes the breaking change version will therefore also likely receive a major version increment.

In this way the version of the Freya meta-package signifies the version of a Freya stack which is known and designed to work well, and it may be many versions ahead of some of the versions of the packages that it depends upon – especially when some of the core packages are very stable.

As from Freya 3.0 (including release candidate builds) the versioning of Freya packages will be less significant from a "feature announcement" perspective, but will become a simple semantic versioning tool.

2.5.1 Upgrading

Upgrading to newer versions of Freya may still result in breaking changes (major version changes will signify this possibility). Guidance for working through these changes can be found in the following section:

- Upgrading – overview and version specific guidance on managing change in Freya versions.

Upgrading

Like any software project, Freya evolves over time. While backwards compatibility is maintained when possible (usually with a deprecation warning to help you to update code to newer constructs), sometimes changing scope or direction requires more significant change.

Guidance for working through specific changes is given in the following sections (this list is likely to expand over time):

2.x → 3.x

This page covers the broad scope of change from 2.x releases of the Freya stack to 3.x releases – including release candidate (RC) builds.

Packages If you are using the Freya meta-package to manage your dependency on Freya, the new packages should be installed correctly on update, especially if you're using Paket.

Hint: In general, Paket comes highly recommended as a reliable way of managing your Nuget – and other – dependencies. For this guide however, standard Nuget tools will work well enough.

Libraries

Arachne In versions prior to 3.x, Freya used the Arachne family of libraries as a type system for the web. As of 3.0, the Arachne libraries have been brought under the Freya stack umbrella and renamed the Freya Types libraries. This requires a change to existing code, with `Arachne.*` being replaced by `Freya.Types.*` – for example `open Arachne.Http` should now be `open Freya.Types.Http`.

Core Various changes to functions have taken place within the Freya Core library, all of which come with backward compatible functions with deprecation warnings. The deprecation messages give suggestions for the function to replace the obsolete function with. Most of these function changes have been about simpliying the optic-based programming model, and separating Pipeline functionality from basic Freya functionality.

Optics The previously provided optics in the Freya Lenses libraries have been renamed and moved to the Freya Optics libraries. Namespaces have changed to reflect this, and `Freya.Lenses.*` has been replaced by `Freya.Optics.*`. For example, `open Freya.Lenses.Http` should now be `open Freya.Optics.Http`.

Routers The router included with Freya prior to 3.x has been moved and renamed to open up the way for additional more specialist routers to become part of the Freya stack. While the basic URI Template based routing has not changed (although the core has been rewritten to be more performant and accurate), the namespace has changed from `Freya.Router` to `Freya.Routers.Uri.Template`. Usage should be unchanged.

The old `Freya.Machine.Router` library to add a `resource` extension to the router is now included in the URI Template Router by default.

Machines As with routers, the machine included with Freya prior to 2.x has been moved and renamed to open up the way for more Machine implementations in future. The machine was previously available in the `Freya.Machine` namespace – this has now been changed to `Freya.Machines.Http`. The requirement to include HTTP functionality through the use of `using http` has been removed.

Machines have gone through a more extensive rewrite than other parts of Freya, and some of the configuration of the HTTP machine has now changed and been simplified. This should all be reflected in deprecated helper methods where applicable, but please do get in Contact if you find areas where this is not the case!

The extension mechanism for machines has also been revamped, as follows:

The CORS support is now available in the namespace `Freya.Machines.Http.Cors`. It can be enabled for a resource by including the keyword `cors` in your machine – `using cors` is no longer needed.

PATCH support is now an extension, and can be found in the namespace `Freya.Machines.Http.Patch`. It can be enabled by adding the `patch` keyword to your machine.

Full details of the new design of HTTP machines can be found in the HTTP machine reference section. Documentation work on Machines is an ongoing process. If the detail you are looking for is not there currently, please check back soon, and follow Freya on Twitter for updates on all Freya subjects, including additions/changes to documentation.

Polyfills Polyfills have been introduced to Freya in 3.x. They allow Freya to work around deficiencies in standards (such as missing data in the OWIN standard) and move faster than current standards allow. They are currently available for Katana based servers and Kestrel. For more information see the reference section on Polyfills.

Errors/Omissions? This document is a work in progress throughout the 3.x release candidate cycle. Please get in Contact if you find any errors or omissions, or if you have any suggestions for how this document can be more useful.

To send a pull request directly, you can use the Edit Page link at the bottom of this (or any other) page.

2.6 Tutorials

For now, see the Getting Started tutorial to create your first Freya application.

Note: More Freya tutorials are currently being developed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

2.7 Getting Started

You can be up and running with Freya in minutes! This quick guide will take you from zero to Hello World, using some of the high level building blocks that Freya provides out of the box – and tell you where to go to learn more about each of them.

2.7.1 Dependencies

You'll be creating a self-contained Hello World implementation as a simple F# console application, so go ahead and create a new empty F# console application. You'll be able to fit the whole thing in the single file that makes up the program (easily!) so there's no need to worry about complex application structures or any of the complexities of frameworks like ASP.NET MVC.

Hint: In general, Paket comes highly recommended as a reliable way of managing your Nuget – and other – dependencies. For this guide however, standard Nuget tools will work well enough.

The first thing you'll need is Freya and a suitable web server. In this case, the simple server from the Katana project will be used, although other servers like Kestrel, IIS, etc. will also work.

Using your preferred method of managing Nuget dependencies, install the required packages. Make sure you're using the latest available versions!

```
PM> Install-Package Freya
PM> Install-Package Microsoft.Owin.SelfHost
```

The Freya package is a meta-package – it brings in all of the packages you'll need for a common Freya application.

2.7.2 Code

At this point you should have an empty F# program. Start off by opening some common namespaces we'll need to write our Hello World program. In this case, you'll need three parts of the Freya stack.

```
open Freya.Core
open Freya.Machines.Http
open Freya.Routers.Uri.Template
```

Greeting

Now you're ready to implement the Freya part of your Hello World application. Start by creating these two functions:

```
let name =
    freya {
        let! name = Freya.Optic.get (Route.atom_ "name")

        match name with
        | Some name -> return name
        | _ -> return "World" }

let hello =
    freya {
        let! name = name

        return Represent.text (sprintf "Hello %s!" name) }
```

You now have two functions which when used together return a representation of `Hello [World|{name}]` depending on whether `{name}` was present in the route. You'll note that these functions are computation expressions – these are very common in Freya and form the basis of the programming model (although computation expression syntax is optional). For more on functions in Freya, see the Core reference. You'll see more about routing in a following section.

Resource

Now you need some way of handling a request and using your `hello` function to return the representation of your greeting as the response – you need a way to model an HTTP resource. You can use the Freya HTTP Machine to do this. Machines are a powerful and high level abstraction – see the Machines reference for more, but for now you can simply use the very simply configured machine below, which will return your representation when a normal "OK" response is valid.

```
let machine =
    freyaMachine {
        handleOk hello }
```

Router

Finally, you'll need a way to make sure that requests to the appropriate path(s) end up at your new Machine-based resource. You can use the URI Template based Freya router to do this easily. The following function will give you a simple router which will route requests matching the given path to your machine. For more on routing in Freya, see the Routers reference.

```
let router =
    freyaRouter {
        resource "/hello{/name}" machine }
```

2.7.3 Server

Now that you have all the "logic" covered you'll need a way of serving it. You can use a simple self-hosted server, and fire it up in the main method of your program. As you're using Katana here, you'll need to create a type of a suitable shape for Katana to use as a start-up object. Here's the code you'll need, along with a main method to start things up.

```
type HelloWorld () =
    member __.Configure () =
        OwinAppFunc.ofFreya (router)

open System
open Microsoft.Owin.Hosting

[<EntryPoint>]
let main _ =

    let _ = WebApp.Start<HelloWorld> ("http://localhost:7000")
    let _ = Console.ReadLine ()

    0
```

And there you have it! Try hitting localhost:7000/hello or localhost:7000/hello/name in a browser – you should have a Hello World up and running.

Hint: The code for the simple Freya Hello World example can be found in the freya-examples GitHub repository here - if you have any problems, try cloning and running the pre-built example.

Hopefully now you're keen to learn more about the Freya components you've seen and what more they can do – and what others are available. The rest of the Freya documentation should help – and if you find it doesn't, please reach out and suggest improvements – Contact is a good place to begin.

2.7.4 Full Code Listing

Here's the complete code for the Hello World program in one place.

```
open Freya.Core
open Freya.Machines.Http
open Freya.Routers.Uri.Template

let name =
    freya {
        let! name = Freya.Optic.get (Route.atom_ "name")

        match name with
        | Some name -> return name
        | _ -> return "World" }

let hello =
    freya {
        let! name = name

        return Represent.text (sprintf "Hello %s!" name) }

let machine =
    freyaMachine {
        handleOk hello }

let router =
    freyaRouter {
        resource "/hello{/name}" machine }

type HelloWorld () =
    member __.Configuration () =
        OwinAppFunc.ofFreya (router)

open System
open Microsoft.Owin.Hosting

[<EntryPoint>]
let main _ =

    let _ = WebApp.Start<HelloWorld> ("http://localhost:7000")
    let _ = Console.ReadLine ()

    0
```

2.8 Recipes

Recipes are availble to help with common patterns of usage, categorised by the various areas of Freya and problems solved. The recipe categories are available directly from the top level navigation.

2.9 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 following two recipe sections:

- 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.

2.9.1 Frameworks

Currently documented integrations:

Suave

Note: This documentation is in the process of being written and reviewed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

2.9.2 Servers

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 issue if they are discovered.

Currently documented integrations:

Katana

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

```
// 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 ()

    0
```

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

Kestrel

Note: This documentation is in the process of being written and reviewed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

2.10 Routing

Routing is a potentially complex topic which also usually conforms to various patterns under normal use. Some of the more common approaches and techniques are defined here as recipes applying to various available Freya routing features.

2.10.1 URI Templates

The URI Template type library, combined with the **Freya.Routers.Uri.Template** library, gives a concise yet powerful approach to routing. URI Templates in routing can sometimes be slightly surprising in their behaviour (the specification has some interesting corner cases) but certain techniques are quite generically useful in routing. Some of those will be added here.

Note: Pull requests suggesting new techniques for routing with URI Templates are very welcome!

Important: The specification of URI Templates is broader than simple path templating and generation, and much of the specification deals with matching query strings, fragments, etc. This means that Freya expects to match a full path and query string. You can make sure this works when you may have an (optional) query string by defining a catch-all query string on the end of your route URI Templates like so: `{?q*}`.

Lists

Matching simple lists of values in URIs can be useful. Using simple URI Template matching, this is trivial:

```
// This uses simple matching and list syntax to match lists of values
// separated by commas

let listTemplate =
    UriTemplate.parse "/{values*}"
```

```
// /one,two,three ->
Freya.Optic.get (Route.list_ "values")
// -> Some [ "one"; "two"; "three" ]
```

Pairs

Matching key/value pairs can also be a useful technique:

```
// This uses simple matching and keys syntax to match lists of key/value
// pairs (x=y) separated by commas

let listTemplate =
    UriTemplate.parse "/{pairs*}"

// /one=a,two=b,three=c ->
Freya.Optic.get (Route.keys_ "pairs")
// -> Some [ ("one", "a"); ("two", "b"); ("three", "c") ]
```

Paths

It is often useful to want to match a whole path, regardless of what it might be – for example, a virtual file server of some kind may map a seemingly "physical" path to a different underlying abstraction.

```
// This uses URI Template path segment matching and list syntax to
// match paths separated by "/" characters

let pathTemplate =
    UriTemplate.parse "{/segments*}"

// /one/two/three ->
Freya.Optic.get (Route.list_ "segments")
// -> Some [ "one"; "two"; "three" ]

let prefixedPathTemplate =
    UriTemplate.parse "/one{/segments*}"

// /one/two/three ->
Freya.Optic.get (Route.list_ "segments")
// -> Some [ "two"; "three" ]
```

2.11 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 in to the main sets of libraries that make up the Freya stack. The library references cover the complete set of Freya functionality, and given both semantic and syntactic reference to the basic usage of the libraries.

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.

2.12 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:

2.12.1 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:

```
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:

```
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:

```
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:

```
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.

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

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

2.12.2 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:

```
// 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.

```
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:

```
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:

```
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>

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

2.12.3 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:

```
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:

```
let yes =
    freya {
        return Next }

let no =
    freya {
        return Halt }

// Both of these functions will be run
Pipeline.compose yes no

// Only the first of these functions will be run
Pipeline.compose no yes
```

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:

```
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.

```
// Types

type Pipeline =
    Freya<PipelineChoice>
```

```
 and PipelineChoice =
    | Next
    | Halt

// Composition

Pipeline.compose : Pipeline -> Pipeline -> Pipeline

// Composition (Operator)

open Freya.Core.Operators

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

2.12.4 Integration

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

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 recipes for integrating with specific servers and frameworks, 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.

```
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.

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

2.13 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.

2.13.1 HTTP

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

- RFC 7230 – Message Syntax and Routing
- RFC 7231 – Semantics and Content
- RFC 7232 – Conditional Requests
- RFC 7233 – Range Requests
- RFC 7234 – Caching
- RFC 7235 – Authentication

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.

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.

To use the types:

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

2.13.2 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.

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.

To use the types:

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

2.13.3 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.

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.

To use the types:

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

2.13.4 PATCH

The **Freya.Types.Patch** library implements types which represent the semantics of the following standard:

- RFC 5789 – PATCH Method for HTTP

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

Note: Full documentation for the individual type designs within Freya.Types.Patch 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.

To use the types:

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

2.13.5 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.

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.

To use the types:

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

2.13.6 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.

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.

To use the types:

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

2.14 Optics

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

2.14.1 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:

```
// 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_`).

2.14.2 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.

```
// 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.accessControlAllowOrigin_`).

2.14.3 HTTP PATCH

The **Freya.Optics.Http.Patch** library provides optics from the `State` to the various aspects of the request and response modelled using the types from **Freya.Types.Http.Patch** (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.

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

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

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

2.15 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.

2.15.1 URI Template

The URI Template router uses URI Templates – supported by the URI Template library – to define and match routes. This is both powerful and useful – the same templates can be used to generate URIs when needed.

Routes

The URI Template router is based on mapping requests to `Pipeline` functions – see Pipelines if you're not familiar with the Freya pipeline concept. It maps the request by matching the method and the path and query against a URI Template.

Syntax

The URI Template Router defines a custom computation expression (simpler and more limited than the `freya` computation expression. The computation expression uses a custom operation to let you define routes in a simple, expressive and typed way. The computation expression is named `freyaUriTemplateRouter` but is also aliased to `freyaRouter` for convenience!

Definitions

Routes are defined by specifying a requirement for the HTTP method (or verb), a requirement for the path, and the `Pipeline` 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 for performance reasons.

Strongly typed values are used for the requirements, using types taken from the Types libraries, in this case **Freya.Types.Http** and **Freya.Types.Uri.Template**, as well as types from **Freya.Routers.Uri.Template**. Here's an annotated example of setting up a router, including opening appropriate modules/namespace:

```
open Freya.Core
open Freya.Routers.Uri.Template
open Freya.Types.Http
open Freya.Types.Uri.Template

// 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 }
```

Breaking this down, the first thing you will notice is the definition of two handlers, which are simple `Pipeline` functions:

```
let handlerA : Pipeline =
    freya {
        return Next }

let handlerB : Pipeline =
    freya {
        return Next }
```

Next in the code you will see two simple URI Templates defined. URI Templates can be used to generate URIs given a template and suitable data – Freya also allows to match on URI Templates, extracting the data to then be used. There are some caveats to matching URI Templates, but in general it is a good fit for many applications.

Two URI Templates are defined which will match the paths shown. The syntax for a match in this case is likely familiar – a simple match, capturing the braced terms (e.g. `{id}`).

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

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

Finally the router itself with the routes defined. The routes are defined using the `route` keyword in the computation expression. This takes three arguments:

- A `UriTemplateRouteMethod`, 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 and query.
- A `Pipeline` which will be called if the method, and the path and query are matched.

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

The router will call the `Pipeline` 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 statically resolved type parameters (don't worry if these are not familiar) to give more concise and flexible APIs. One of the places where this is used is in this computation expression. Rather than only taking a literal `UriTemplateRouteMethod` as the first parameter, the `route` function can actually take any value which has a static `UriTemplateRouteMethod` member. By default, this is defined for a few different types, all of which can be used interchangeably. That means that the folloing are all valid routes:

```
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
```

You will see that this potentially makes things clearer and more readable, allowing for simpler expressions of the same concepts. In addition to the Methods, both the template and the handler are also inferred - anything which has `UriTemplate` and anything which has `Pipeline` 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:

```
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+ – the router implements `Pipeline` and thus anything which expects to be able to infer a pipeline can accept a router.

URI Templates

in this example the URI Templates have been defined separately from the router. This could be done 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 they are commonly defined 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.

Values

As seen in Routes, it's quite simple to map routes (the combination of method and path specification) to `Pipeline` functions. Once a route is matched however, you will likely need the handler to have access to the data that was matched as part of the URI Template.

Here is one of the URI Templates from the previous section:

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

You can see that if this route has been matched, a value for `{id}` should now exist. Freya aims for a consistent model of programming throughout, and so the approach to accessing this data is aligned to accessing any other data – it is considered to be part of the state.

Optics

As with accessing data from the request or response, accessing data from the route requires the use of a suitable set of optics. These are provided under the `Route` module. Here's a function which will return the `{id}` value from the example:

```
let readId =
    freya {
        return! Freya.Optic.get (Route.atom_ "id") }
```

There are several things to note here. The first is that route optics are prisms. You can't statically be sure that a value will be present (even though intuitively you can know that it will be in certain contexts), so the optics must be prisms. Additionally, route optics are parameterised – as shown, it is passed the name of the value sought, in this case "id").

Another key point to note is that there are three prisms available (all within the `Route` module). In this case, `Route.atom_` is used, but `Route.list_` and `Route.keys_` are also available. Briefly, this is due to the nature of data within the URI Template specification. It is possible to render and match more complex data structures with URI Templates (see the URI Template RFC for a sense of how URI Templates can be used in more advanced ways). Simple values will only usually require the use of the `Route.atom_` optic, but much more is possible – see the relevant recipes for more information.

Summary

Techniques for accessing matched values using optics have been shown.

```
open Freya.Routers.Uri.Template

// Optic for extracting string values from a matched route
Route.atom_ : string -> Prism<State, string>

// Optic for extracting string list values from a matched route
Route.list_ : string -> Prism<State, string list>

// Optic for extracting string pair values from a matched route
Route.keys_ : string -> Prism<State, (string * string) list>
```

Recipes

See also:

- Routing – as well as the library reference, a growing collection of routing recipes covering various techniques is maintained.

See also:

- Routing – as well as the library reference, a growing collection of routing recipes covering various techniques is maintained.

2.16 Machines

Freya provides powerful ways to interact with the web which are safe, expressive, and relatively low-level. However, Freya also provides some higher level abstractions to enable more effective and concise programming when interacting with complex standards.

Machines are one way of approaching this, and provide a different model of web programming, based around decision trees and declarative programming (don't worry – this sounds complex and scary, but it isn't!)

Understanding the way that Machines are built and used helps make the most of this powerful Freya feature, and the underlying approach and computational model is covered in some depth in the topic: Machines – design and implementation of a theoretical Machine.

Freya currently includes one machine (with extensions) for working with HTTP – additions may be made in future for complementary protocols, etc.

- HTTP – a Freya Machine for working with HTTP in a safe and semantically correct way, based on a purely functional declarative approach.

In addition to the documentation on specific Machines, all Freya Machines follow a defined set of Conventions which are useful to note.

2.16.1 HTTP

Freya lets you work with HTTP implicitly, using the typed but low-level tools found in Core combined with types and optics, such as the provided Optics. These are expressive and effective, but provide little in the way of structure.

HTTP is a fairly involved set of standards, and properly implementing the correct semantics of HTTP is not a trivial matter. Dealing with the correct logical approach to negotiating content types, working out the right approach to cache control and correctly expressing the state of the underlying resources, can be quite a lot to design when taken holistically.

Freya.Machines.Http is designed to solve this. The Machine lets you define just the properties of a resource you care about, and the Machine library handles the rest. You can specify as much or as little as you wish, all in a type safe manner, and be confident that HTTP semantics will remain consistent and correct.

As noted in the topic on the design of Machines, configuration of a machine is the key step in effective use. The HTTP Machine has extensive configuration options.

- Decisions – decisions influence the overall execution of the HTTP Machine, and enable you to define the behaviour of key aspects of your resource by answering true/false questions about the resource.
- Handlers – handlers enable the response to return representations of the resource when appropriate, as well as providing a potential point in the execution of the Machine to set additional headers, etc.

Decisions

Note: This documentation is currently being written and reviewed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

Handlers

Important: This documentation is currently partially complete and is in the process of being written and reviewed. Please check back soon. Documentation updates are also announced on the Freya Twitter feed – follow Freya there for up to the minute information on changes to this documentation, and other Freya news.

Handlers are called as the final step in an HTTP Machine execution – they are used to set properties of the response, usually a payload (where applicable) and potentially additional headers. Handlers are always optional – when a handler is not defined, the response will simply be returned without a payload being set (and associated headers). This is a valid approach in many situations, especially in HTTP cases where a payload is not common for the response.

Representations

The HTTP Machine allows you to declare handlers in three ways, allowing you to use a simpler formulation when it is all that's required. All handlers must return a `Representation`, which defines the content and associated metadata (such as media type, language, etc. where relevant), but this may be done in the following ways:

Dynamic Negotiated The most comprehensive case allows for the selection of a representation according to the content negotiation which the HTTP Machine may have performed (if you have configured the Machine to do so – see the section on Properties). In this case, the handler defined within the Machine must be of the form `Acceptable -> Freya<Representation>`, where `Acceptable` is a type giving the results of content negotiation. The handler may use this information to select the most appropriate representation.

Dynamic A simpler case occurs when content negotiation is not realistically required – this may often be the case for simpler APIs, or where the API only offers limited options (an all JSON API for example). In this case the handler must be of the form `Freya<Representation>` – the handler may work with the request, response, and other data, but is not given information regarding content negotiation.

Static The simplest case is for instances when the representation is known at compile time, and a handler of the form `Representation` can be defined. This is useful for simple static content, etc.

Status Codes

Handlers are defined and named according to the following tables, categorised by response status code (these codes, along with other commonly recommended/required headers will be set automatically). Note that some status codes may be returned by more than one handler, such as the handlers for a normal **200** response for OPTIONS requests, and for other requests.

2.16.2 Conventions

Freya Machines follow a package naming convention to make it clear what is available and how it should be used. The convention for naming is as follows:

- **Freya.Machines.<Machine>[.<Extension>]**

Using the example of the Freya HTTP Machine, this gives:

- **Freya.Machines.Http**

as the name of the core HTTP Machine library. The HTTP Machine also has extensions available to implement further web standards, or to integrate with other technologies/frameworks. Following the convention, this gives (for example):

- **Freya.Machines.Http.Cors**
- **Freya.Machines.Http.Patch**

2.17 Polyfills

Polyfills provide functionality which is not (and may never be) part of an open standard. Existing polyfills add existing data to the OWIN standard, enabling such things as more accurate routing. Polyfills are available for the following servers:

- Katana – polyfills for Katana-based implementations, including self-hosted and IIS hosted approaches.
- Kestrel – polyfills for Kestrel-based servers.

2.17.1 Katana

The Katana framework is the basis of self-host OWIN applications and also IIS applications. The **Freya.Polyfills.Katana** polyfill should be used whenever you are taking one of these hosting approaches. The polyfill should be used by inserting it in front of your Freya application in a pipeline composition:

```
open Freya.Core
open Freya.Core.Operators
open Freya.Polyfills.Katana

// Some pre-existing Freya pipeline (could be a router, function, etc.)
let myApp =
    freya {
        ...
        return Next }

// Compose a polyfill with the pre-existing pipeline
let composedApp =
        Polyfill.katana
    >?= myApp

// Use as normal...
let owinApp =
    OwinAppFunc.ofFreya composedApp

...
```

The polyfill currently adds additional data to the OWIN environment which enables routing to work more accurately (giving routers access to the raw, encoded form of the path and query).

2.17.2 Kestrel

The Kestrel server is the newer HTTP server built by Microsoft. The **Freya.Polyfills.Kestrel** polyfill should be used whenever you are taking one of these hosting approaches. The polyfill should be used by inserting it in front of your Freya application in a pipeline composition:

```
open Freya.Core
open Freya.Core.Operators
open Freya.Polyfills.Kestrel

// Some pre-existing Freya pipeline (could be a router, function, etc.)
let myApp =
    freya {
        ...
        return Next }
```

```
// Compose a polyfill with the pre-existing pipeline
let composedApp =
        Polyfill.kestrel
    >?= myApp

// Use as normal...
let owinApp =
    OwinAppFunc.ofFreya composedApp

...
```

The polyfill currently adds additional data to the OWIN environment which enables routing to work more accurately (giving routers access to the raw, encoded form of the path and query).