Tooling Tutorial

Overview

Ballerina tooling currently includes a standalone IDE called Ballerina Composer and also offers IDE plugins for Visual Studio Code and IntelliJ IDEA. In addition to IDE tools, there are a few command line tools available to help with package management, tests, documentation generation and Swagger/OpenAPI support.

Ballerina Composer

Ballerina Composer is the standalone IDE shipped with Ballerina installer. It will get installed alongside Ballerina installation. Please download the specific installer for your platform from ballerina.io and install it to get Ballerina Composer up and running.

To Launch Ballerina Composer after installation,

- Windows : Search for Ballerina Composer in start menu
- Mac: Search for Ballerina Composer in Applications folder
- Linux: Execute 'composer' command in the terminal

IDE plugins

We currently provide plugins for Visual Studio Code and IntelliJ IDEA. To install them, please follow below instructions.

Visual Studio Code (VSCode)

- Go to *view* -> *Extensions* and search for "Ballerina"
- Plugin details page show in-detail instructions for setting up.
- As explained in plugin description, set **`ballerina.home`** config and point it to Ballerina installation director.

IntelliJ IDEA (IDEA)

- Go to *Preferences* -> *Plugins* and type "Ballerina" in search box. Then click on "search in repositories"

Only stable releases will be available via Plugins registry. To install other versions (eg: nightlies), please download the plugin from www.ballerina.io/downloads and use the option to install from file.

Which one should I use?

If you are already familiar with either VSCode or IDEA, we would recommend trying the specific plugin for it. Otherwise, below is a feature comparison between different tools.

If you are still in doubt, we would recommend using VSCode and the Ballerina plugin for it. Even Though it doesn't offer all the features available in Ballerina Composer, it offers a simple and lightweight experience with all the mandatory language features, plus a nice preview of the Ballerina Program in sequence diagrams.

Feature	Ballerina Composer	VSCode Plugin	IDEA Plugin
Basic Language Support (syntax highlighting, diagnostics, refactoring support, etc.)	✓	✓	✓
Diagram View	✓	✓	✗
Diagram Editing	✓	✗	✗
Debugging	✓	✓	✓
TryIt Tool	✓	✗	✗
Trace Logs View	✓	✗	✗

How to start writing a program?

Now that you have chosen your IDE, let's proceed with creation of a project.

- In your terminal, goto the directory you want the project to be created in and execute **`ballerina init`** command to create a new Ballerina project. `init` command will create the initial Ballerina project structure as below.

```
MacBook-Pro:demo kavith$ mkdir hello_project
MacBook-Pro:demo kavith$ cd hello_project/
MacBook-Pro:hello_project kavith$ ballerina init
Ballerina project initialized

MacBook-Pro:hello_project kavith$ tree
.
├── Ballerina.toml
└── hello_service.bal

0 directories, 2 files
```

- **`ballerina init -i`** lets you to customize the project generation wizard. It allows you to generate Ballerina services/test skeletons along with packages to start development right away.

```
MacBook-Pro:hello-service kavith$ ballerina init -i
Create Ballerina.toml [yes/y, no/n]: (y) y
Organization name: (kavith) kavith-demo
Version: (0.0.1)
Ballerina source [service/s, main/m, finish/f]: (f) s
Package for the service : (no package) demo
Ballerina source [service/s, main/m, finish/f]: (f) f
Ballerina project initialized

MacBook-Pro:hello-service kavith$ tree
.
├── Ballerina.toml
└── demo
    ├── Package.md
    ├── hello_service.bal
    └── tests
        └── hello_service_test.bal

2 directories, 4 files
```

Project Structure

```
/
  .gitignore
  Ballerina-lock.toml  # Generated during build, used to rebuild identical binary
  Ballerina.toml       # Configuration that defines project intent
  .ballerina/          # Internal cache management and contains project repository
                       # Project repository is built or downloaded package dependencies

  main.bal             # Part of the “unnamed” package, compiled into a main.balx
                       # You can have many files in the "unnamed" package, though unadvisable

  package1/            # The source in this directory will be named “<org-name>/package1”
    Package.md         # Optional, contains descriptive metadata for display at Ballerina Central
    *.bal              # In this dir and recursively in subdirs except tests/ and resources/
    [tests/]           # Package-specific unit and integration tests
    [resources/]       # Package-specific resources

  packages.can.include.dots.inthe.dir.name/
    Package.md
    *.bal
    [tests/]
    [resources/]

  target/              # Compiled binaries and other artifacts end up here
      main.balx
```

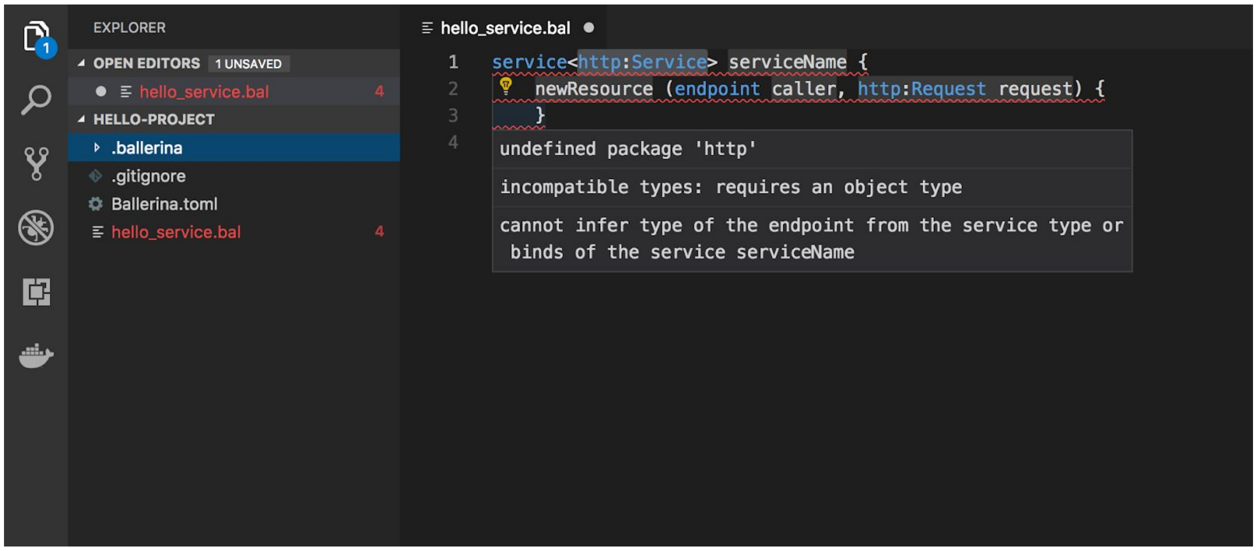
For more information on the structure of Ballerina projects, please refer to the guide available in ballerina.io.

- Open the created project in your IDE (This tutorial assumes you are using VSCode) and open *hello_service.bal* file in the editor. This file will by default contain a hello world service created by ballerina init command.
- Lets delete the content in *hello_service.bal* and create from the scratch while exploring the set of features offered to make development easy.
- You can observe that semantic and syntactic errors are shown (diagnostics) as we type and also observe how the code auto-completion and suggestions are provided according to current scope.

Intellisense and Diagnostic Capabilities

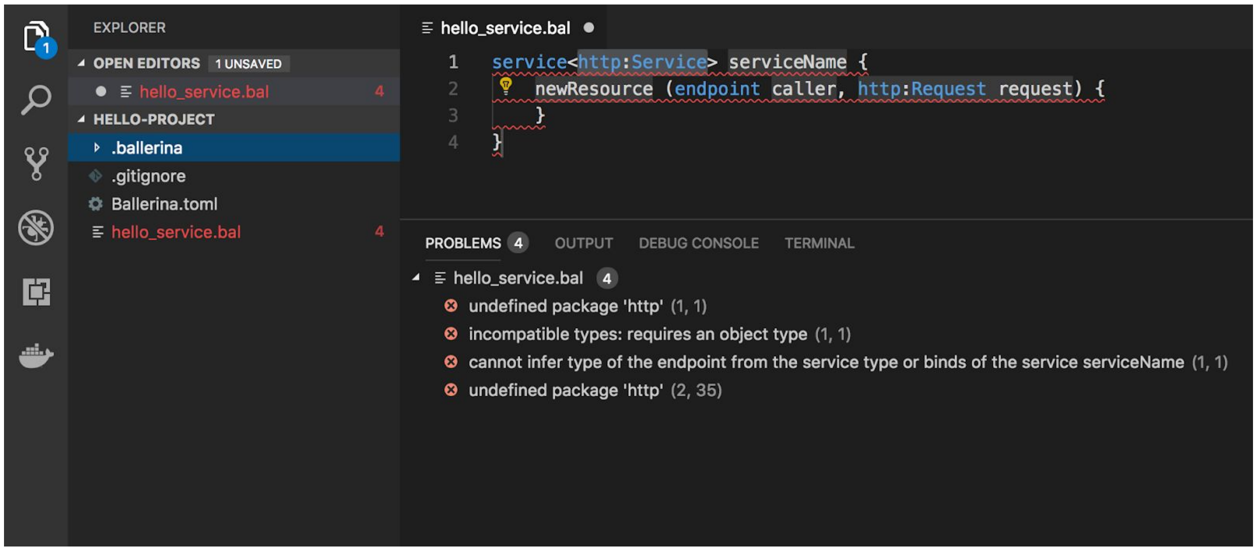
Ballerina Language server is the component which is providing the language smartness to the IDE plugin. Ballerina Language Server supports the following Intellisense capabilities.

Diagnostics



```
service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {
    }
```

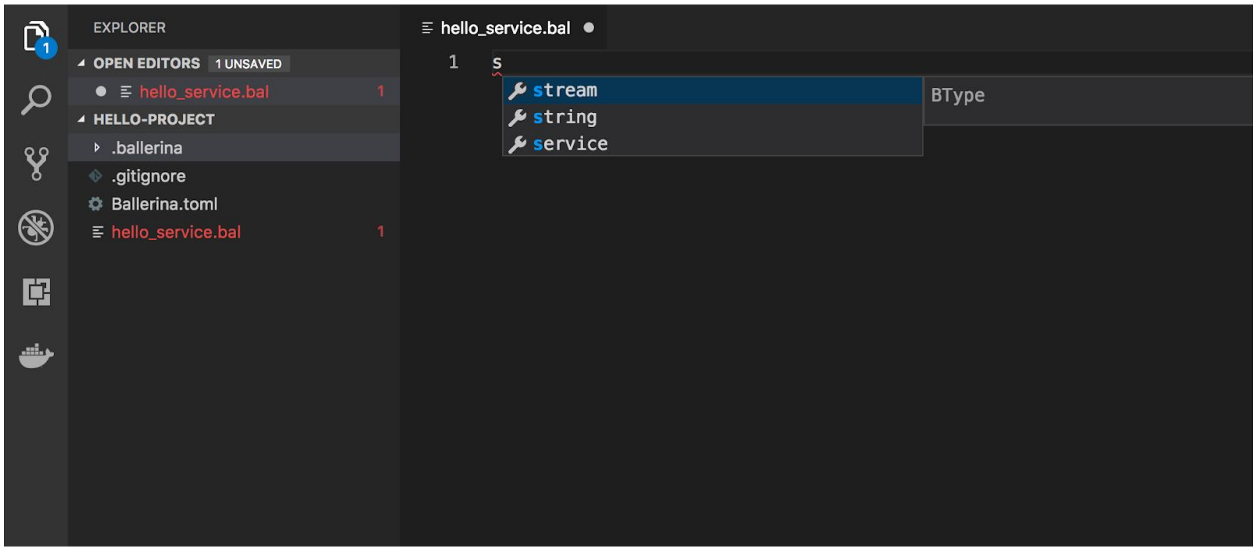
```
undefined package 'http'
incompatible types: requires an object type
cannot infer type of the endpoint from the service type or
 binds of the service serviceName
```



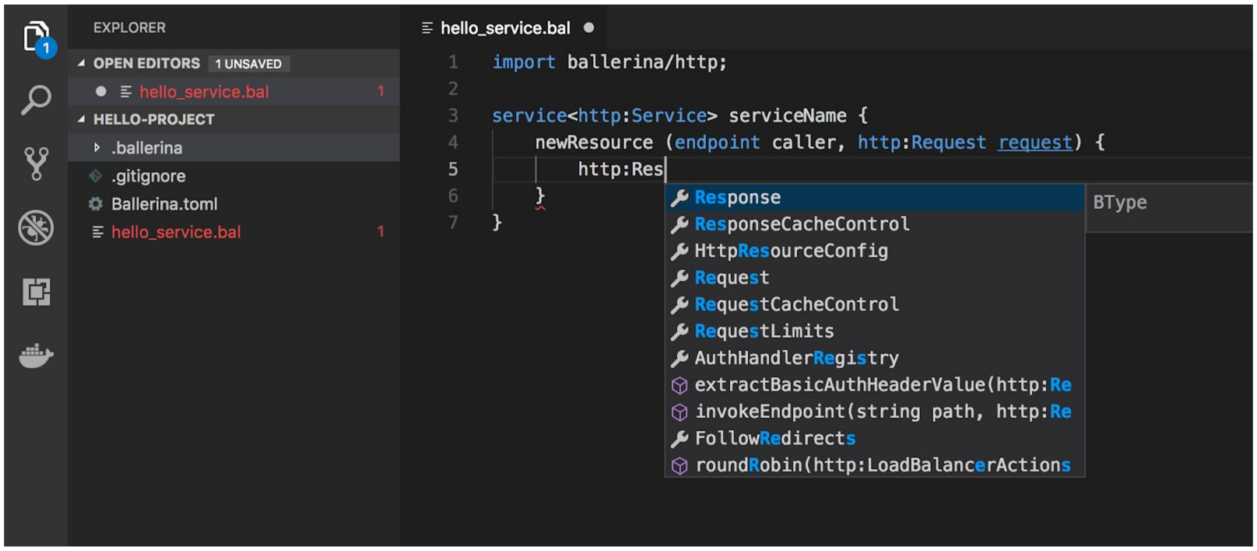
```
service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {
    }
}
```

```
hello_service.bal 4
  undefined package 'http' (1, 1)
  incompatible types: requires an object type (1, 1)
  cannot infer type of the endpoint from the service type or binds of the service serviceName (1, 1)
  undefined package 'http' (2, 35)
```

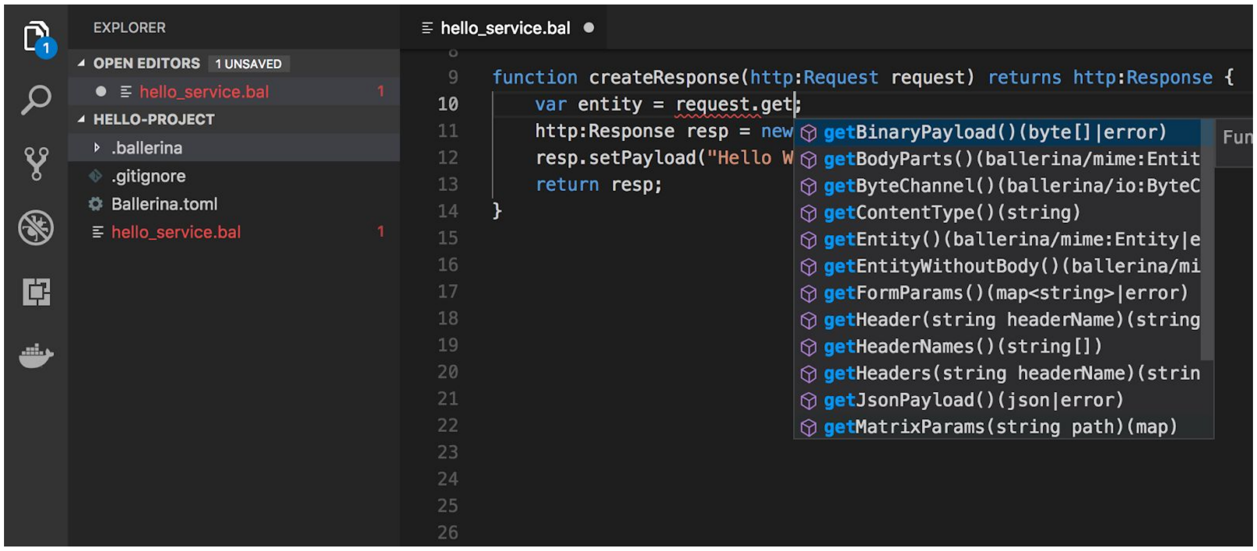
Completions



```
s
```



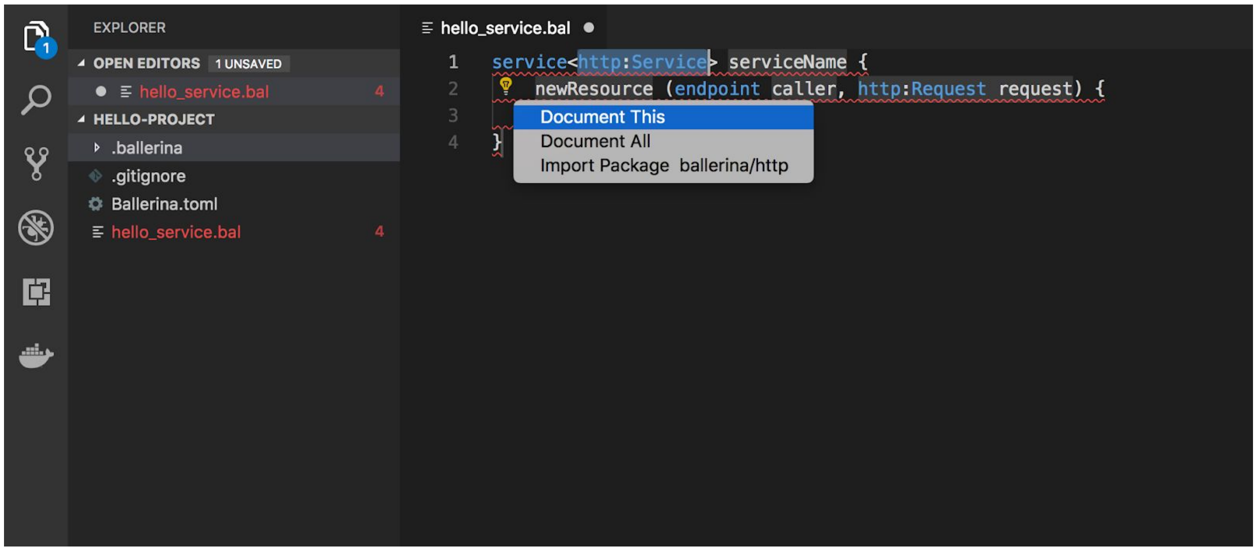
```
import ballerina/http;

service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {
        http:Res
    }
}
```


EXPLORER
OPEN EDITORS 1 UNSAVED
hello_service.bal 1
HELLO-PROJECT
.ballerina
.gitignore
Ballerina.toml
hello_service.bal 1
hello_service.bal
getBinaryPayload()(byte[]|error)
getBodyParts()(ballerina/mime:Entit
getByteChannel()(ballerina/io:ByteC
getContentType()(string)
getEntity()(ballerina/mime:Entity|e
getEntityWithoutBody()(ballerina/mi
getFormParams()(map<string>|error)
getHeader(string headerName)(string
getHeaderNames()(string[])
getHeaders(string headerName)(strin
getJsonPayload()(json|error)
getMatrixParams(string path)(map)
Fun

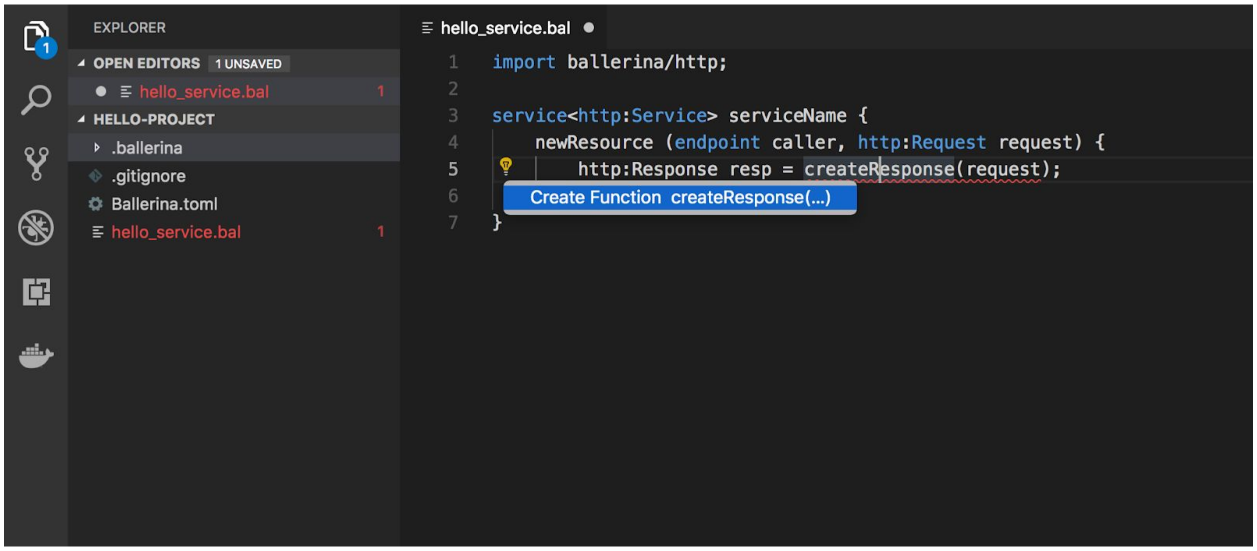
```
function createResponse(http:Request request) returns http:Response {
    var entity = request.get;
    http:Response resp = new
    resp.setPayload("Hello W
    return resp;
}
```

Refactor support

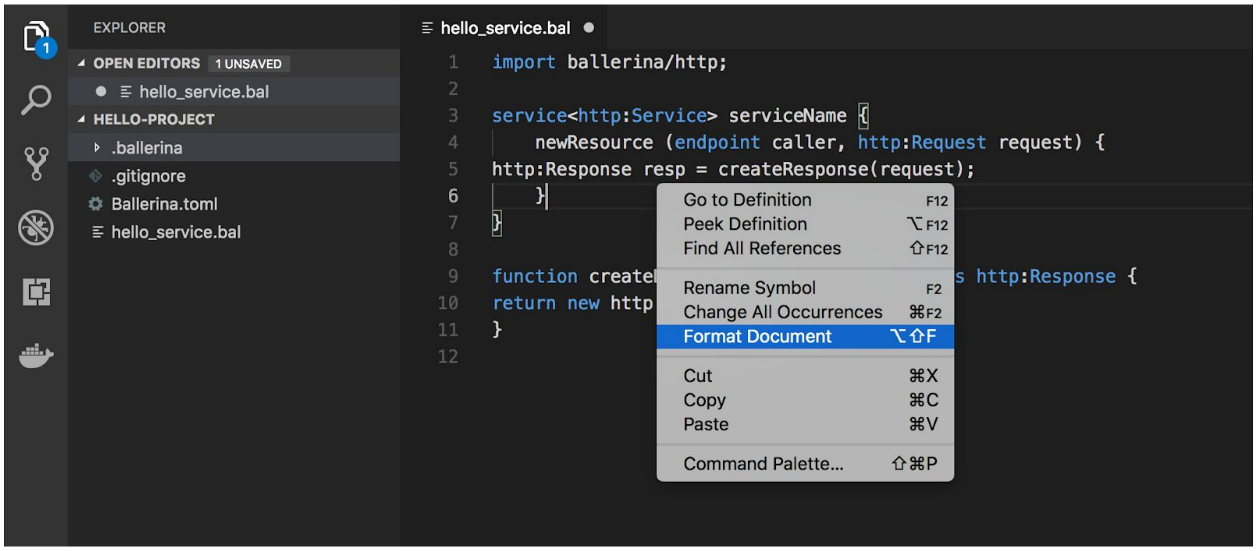
Code Actions, Rename, Add Imports, Add Documentation, Add Undefined Functions, Code Formatting



```
service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {

}
```

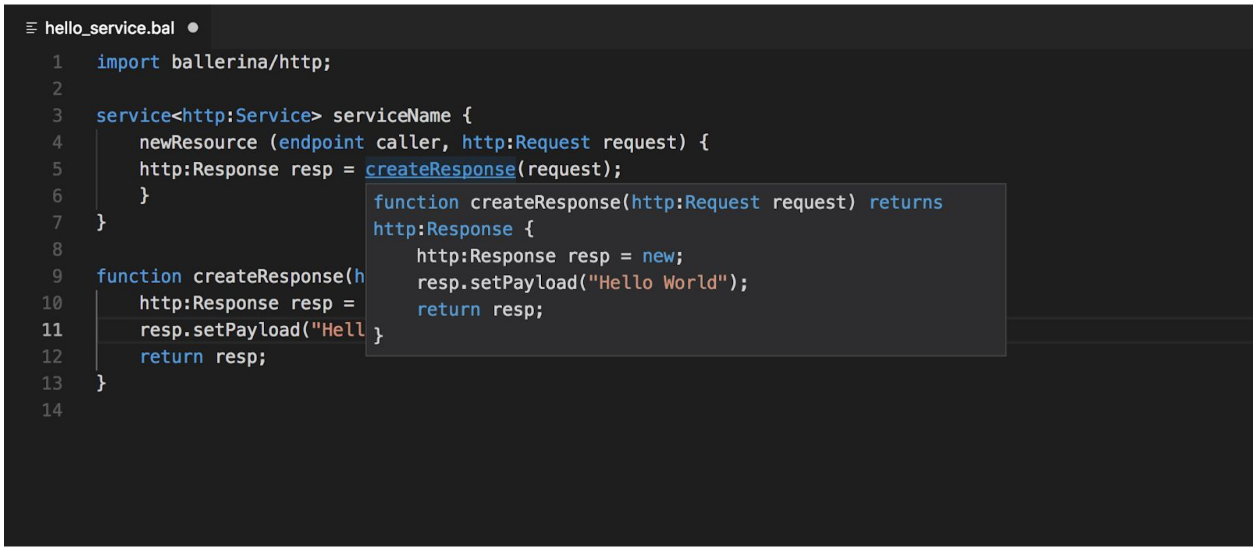


```
import ballerina/http;

service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {
        http:Response resp = createResponse(request);

}
```



```
import ballerina/http;

service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {
http:Response resp = createResponse(request);
    }
}

function create                          s http:Response {
return new http
}
```

Go to Definition

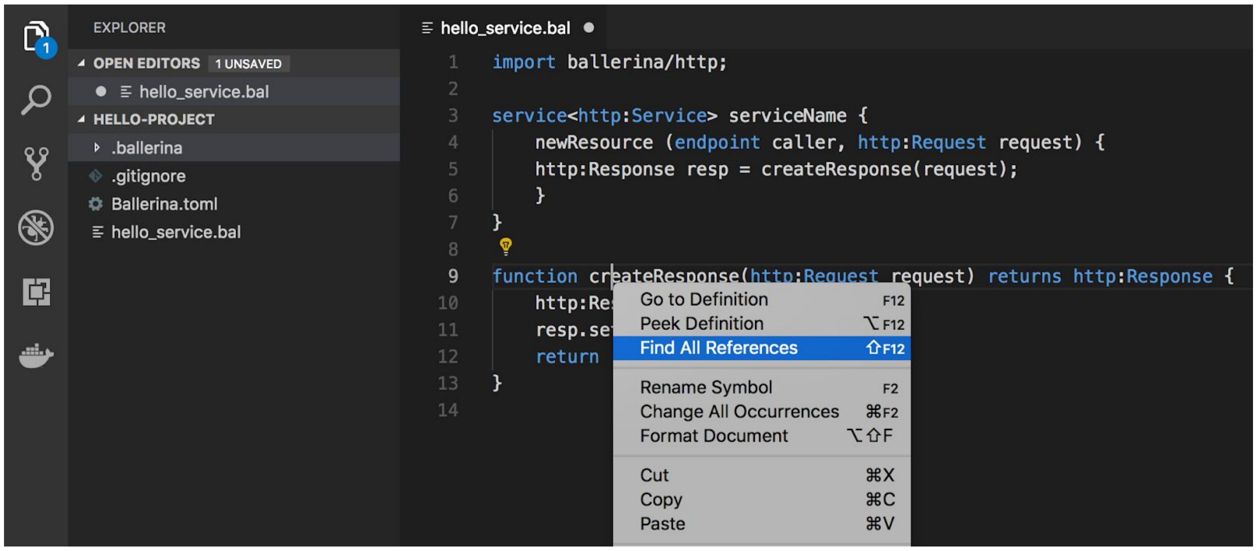


hello_service.bal

```
import ballerina/http;

service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {
    http:Response resp = createResponse(request);
    }
}

function createResponse(h
    http:Response resp =
    resp.setPayload("Hell
    return resp;
}
```

```
function createResponse(http:Request request) returns
http:Response {
    http:Response resp = new;
    resp.setPayload("Hello World");
    return resp;
}
```

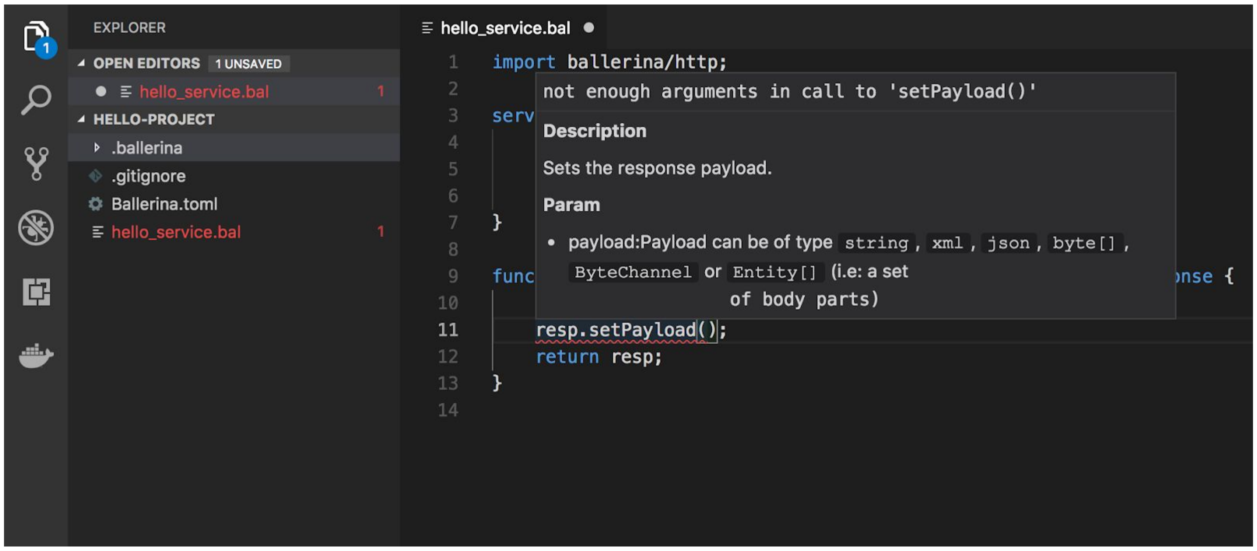
Find References



hello_service.bal

```
import ballerina/http;

service<http:Service> serviceName {
    newResource (endpoint caller, http:Request request) {
    http:Response resp = createResponse(request);
    }
}

function createResponse(http:Request request) returns http:Response {
    http:Re
    resp.se
    return
}
```

Signature Help



```
import ballerina/http;

serv

}

func                                                               nse {
    resp.setPayload();
    return resp;
}
```

```
not enough arguments in call to 'setPayload()'
```

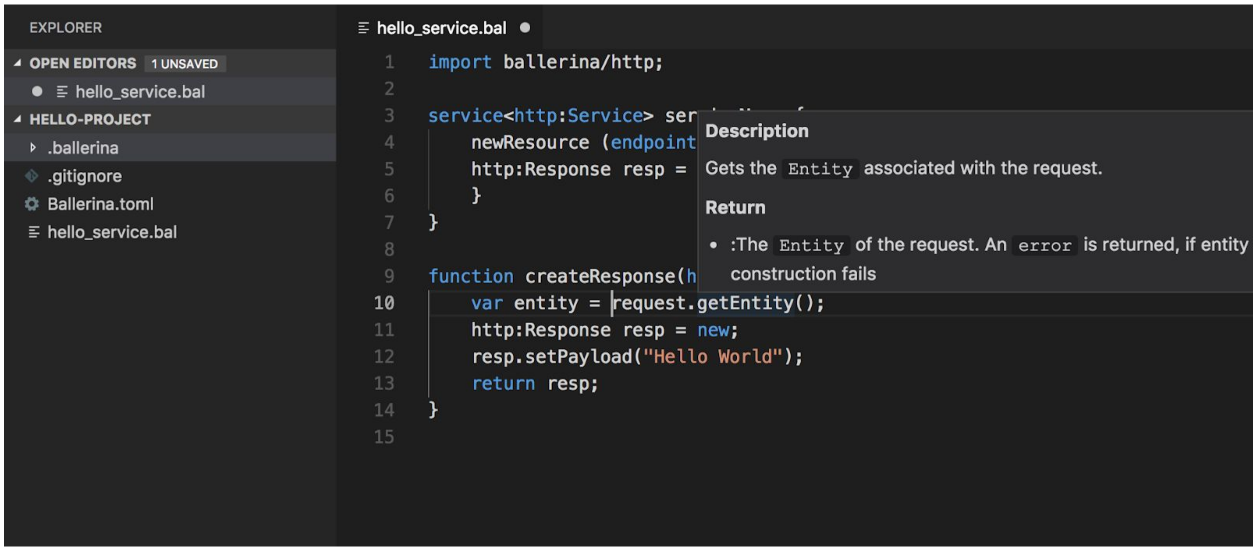
Description

Sets the response payload.

Param

- payload:Payload can be of type `string`, `xml`, `json`, `byte[]`, `ByteChannel` or `Entity[]` (i.e: a set of body parts)

Hover Support



```
import ballerina/http;

service<http:Service> ser
    newResource (endpoint
    http:Response resp =
    }
}

function createResponse(h
    var entity = request.getEntity();
    http:Response resp = new;
    resp.setPayload("Hello World");
    return resp;
}
```

Description

Gets the `Entity` associated with the request.

Return

- :The `Entity` of the request. An `error` is returned, if entity construction fails

Testing Ballerina Programs

- Now let’s write a unit test for a Ballerina service.
- A test skeleton is generated with the ballerina init -i command.
- To run tests for a package execute command `ballerina test <package-name>`

```
my_project
├── Ballerina.toml
├── hello_service.bal
├── my_package
              ├── hello_service.bal
              ├── tests
                          ├── test_hello_service.bal
```

For more info on testing Ballerina code, please refer to the guide available in ballerina.io.

API Doc Generation

- You can use the doc generator tool to generate API Docs for your Projects. Ballerina Documentation is part of the Language Syntax itself and Documentation Block Support markdown syntax with some additional tags to refer parameters and attributes.
- Now let’s generate API Docs for your package. Go to your example project root and enter following command **`ballerina doc <package_name>`**
- If you need to generate API Docs for the whole project use the following command `ballerina doc`
- Now go to the target directory in your project and you can find the generate API Docs under api-docs directory

For more info on documenting Ballerina code, please refer to the guide available in ballerina.io.

Package Management

Through Ballerina Central (https://central.ballerina.io) you can share your packages with Ballerina Community and pull packages shared by others.

Setting Up for pushing

- Login to Ballerina Central and create an organization. Now you can get the access token from https://central.ballerina.io/dashboard .
- Get your access token and add this to **`Settings.toml`** `(<USER_HOME/.ballerina)`

Pushing a package

- Now Let's publish your package to Ballerina Central. Go to Your project and open Ballerina.toml and change the org-name config accordingly.
- Now go to your project root and build your package with following Command
  `ballerina build <your_package_name>`
- This will build your package and now you can publish your package to Ballerina Central with the following Command
  `ballerina push <your_package_name>`
- Go to Ballerina Central and you can see the published package

Pulling Packages

- If you need to use an already published package to Ballerina Central, you can pull a package.
- In order to pull a package from central with the following commands
  `ballerina pull <org_name>/<package_name>:<version>` (Pull a specific version)
- **`ballerina pull <org_name>/<package_name>`** (Pull the latest version)

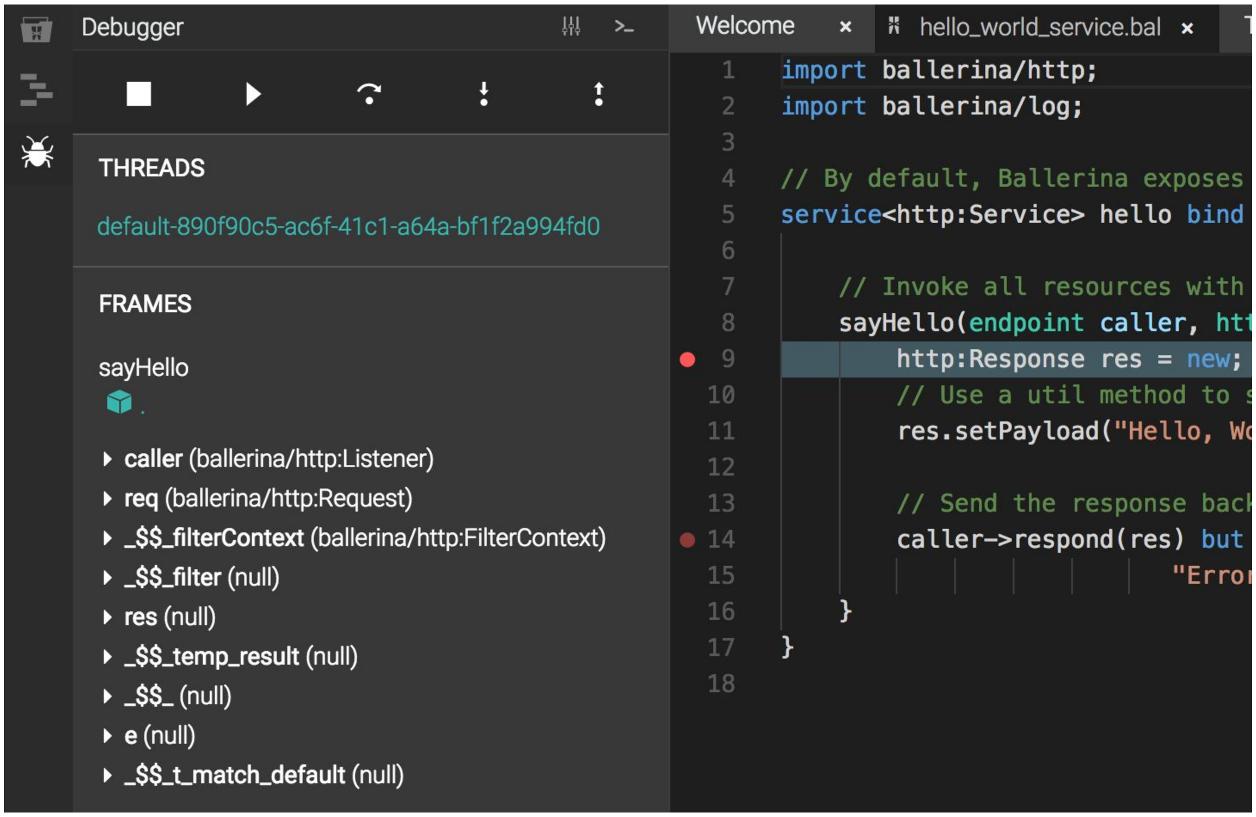
For more info on package management in Ballerina, please refer to the guide available in ballerina.io.

Exploring Ballerina Composer

Now let's switch over to Ballerina Composer and explore several features it provides.

Debugging Ballerina Programs

- Open your Ballerina Project in Ballerina Composer (File->Open Project).
- Now add necessary debug points to your source and open the debug panel at left. Click on the Debug option to start debugging your service.
- Now invoke the service and observe how the debug points are hit and the snapshots of the variables and their values.



default-890f90c5-ac6f-41c1-a64a-bf1f2a994fd0

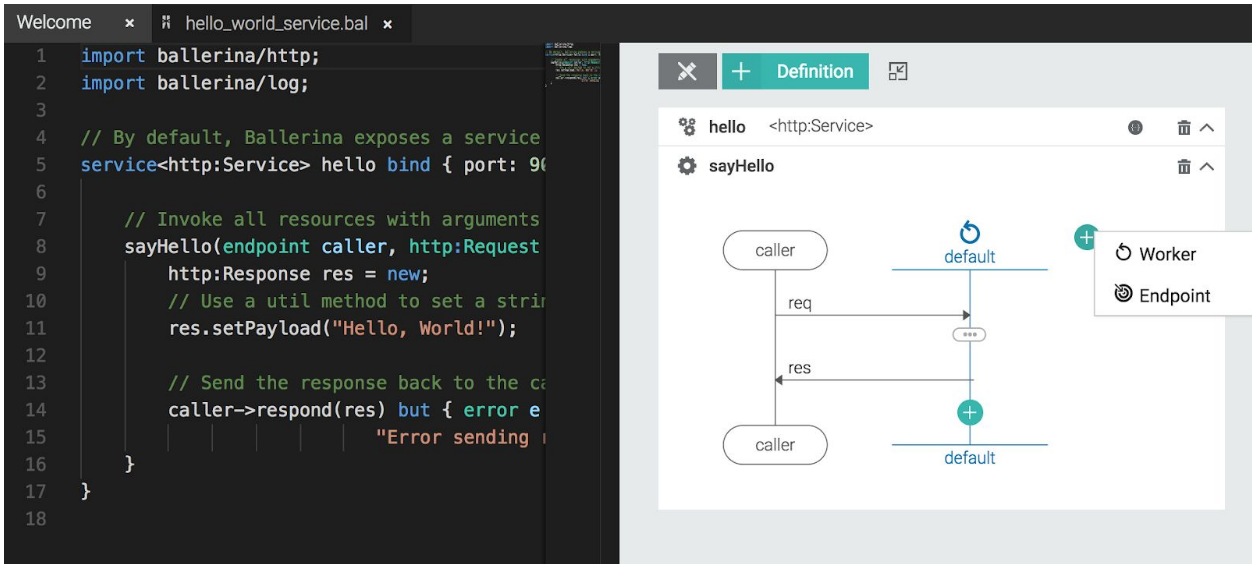
sayHello

- caller (ballerina/http:Listener)
- req (ballerina/http:Request)
- _$$_filterContext (ballerina/http:FilterContext)
- _$$_filter (null)
- res (null)
- _$$_temp_result (null)
- _$$_ (null)
- e (null)
- _$$_t_match_default (null)

```
import ballerina/http;
import ballerina/log;

// By default, Ballerina exposes
service<http:Service> hello bind

    // Invoke all resources with
    sayHello(endpoint caller, htt
        http:Response res = new;
        // Use a util method to s
        res.setPayload("Hello, Wo

        // Send the response back
        caller->respond(res) but
                            "Error
    }
}
```

Design View

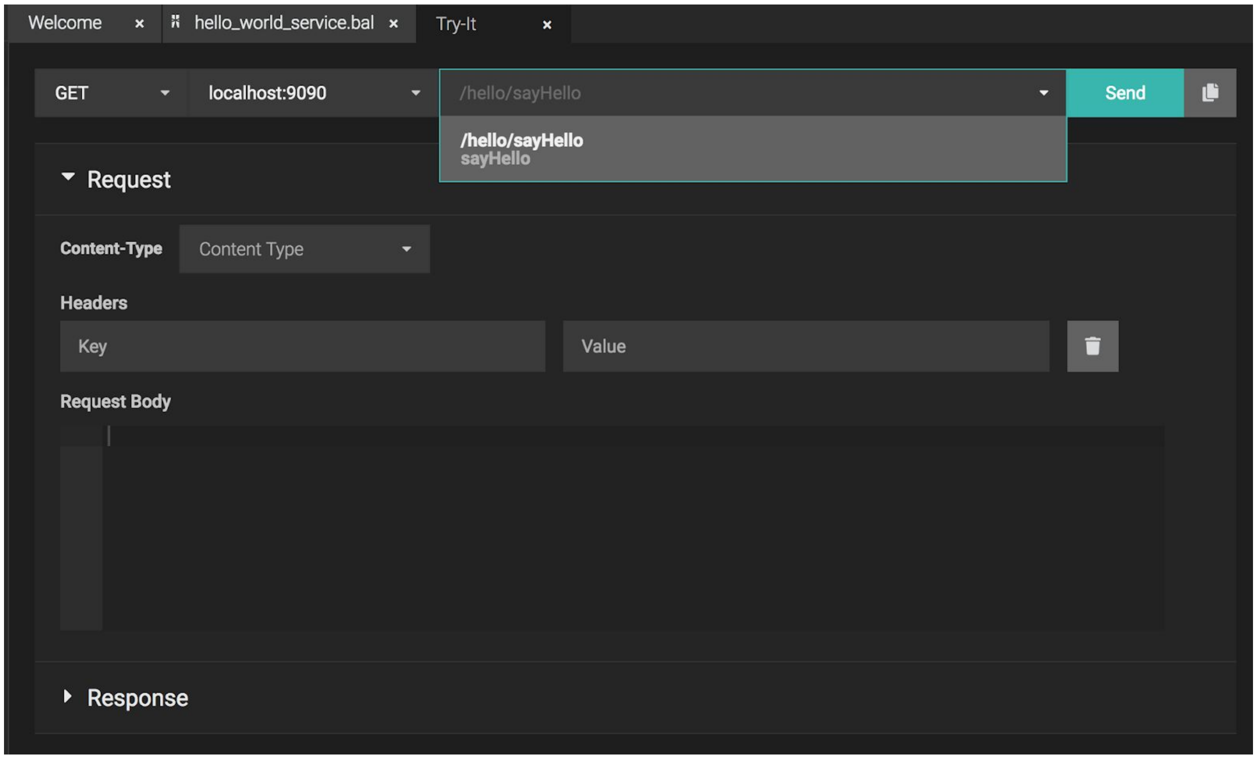
- Ballerina has the visual parity for your Ballerina programs. Open your `hello_service.bal` and click on the design view Icon in composer at the right bottom corner. You can observe how the invocations are shown in a sequence diagrammatic manner.
- You can benefit with the visual editing support, for creating your program skeletons.
- Click on the Edit option on the design view and add definition skeletons from the design view. In the split view, you can see when you add a new construct from the design, respective ballerina source is generated as well.



```
import ballerina/http;
import ballerina/log;

// By default, Ballerina exposes a service
service<http:Service> hello bind { port: 9(

    // Invoke all resources with arguments
    sayHello(endpoint caller, http:Request
        http:Response res = new;
        // Use a util method to set a strir
        res.setPayload("Hello, World!");

        // Send the response back to the ca
        caller->respond(res) but { error e
                                "Error sending r
    }
}
```

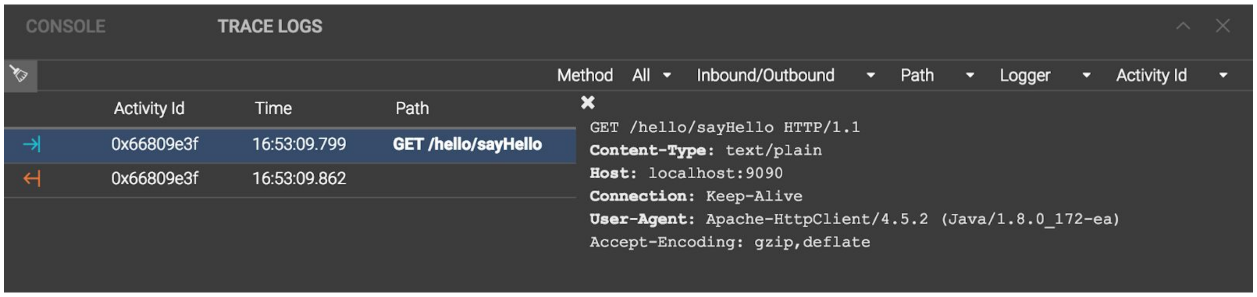
Try It

- In the Composer you can invoke your services with the available Try It option. In the debug panel you can find the run option for your Ballerina Programs.
- Open your `hello_service.bal` and click on the run item. In the Composer Console, you will be prompted with the Try It.
- Now let's click on the Try It and let's observe how to invoke your service. You can find the available resources in the drop down and select the desired resource and invoke it with the sample data.



Trace Logs

- When invoking the services, you can see the Trace Logs for the services through the Trace Log Console at the bottom.
- Invoke the service from Try It and click on the Trace Log option next to Console. You can see the inbound and outbound Trace Logs for your service invocation.



	Activity Id	Time	Path
→|	0x66809e3f	16:53:09.799	GET /hello/sayHello
←|	0x66809e3f	16:53:09.862	

```
GET /hello/sayHello HTTP/1.1
Content-Type: text/plain
Host: localhost:9090
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.5.2 (Java/1.8.0_172-ea)
Accept-Encoding: gzip,deflate
```

Swagger OpenAPI Support

- Ballerina allows you to generate a service stub or a client connector from a swagger or OpenAPI file.

```
ballerina swagger [mock|client] <swagger_file>
[-p <packagename> | --package <packagename>]
[-o <path> | --output<path>]
```

- Similarly you can generate a swagger or OpenAPI out from a Ballerina Service Source file.

```
ballerina swagger export <balfile>
[-o <path> | --output <path>]
[-s <servicename> | --service <servicename>]
```

- Ballerina Composer allows you to edit an HTTP Service definition in a swagger view.