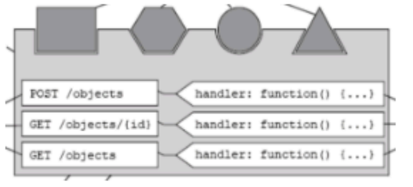
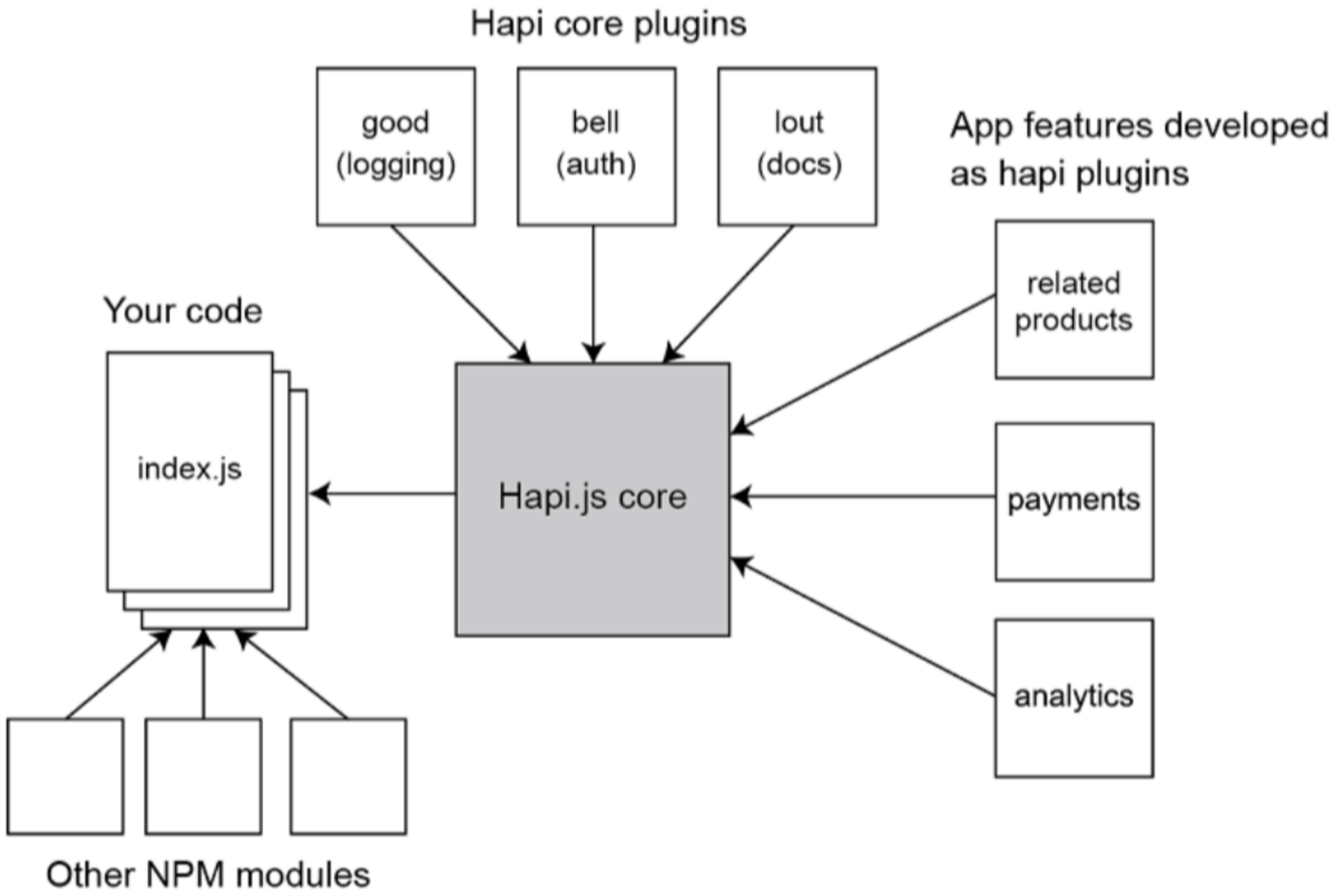
HAPI Building Blocks

Hapi Building Blocks



Enumerate the core building blocks of hapi and explain how these are assembled into a simple application.

Example Hapi Application Structure



Convention over Configuration

I LOVE TO WRITE A BUNCH OF CONFIGURATION FILES BEFORE WRITING ACTUAL CODE

- Said no one ever

Reasonable defaults
Only specify the unconventional bits
Reduce number of decisions to be made
Eliminate distractions

Convention over Configuration

Convention over configuration (also known as **coding by convention**) is a software design paradigm used by software frameworks that attempt to decrease the number of decisions that a developer using the framework is required to make without necessarily losing flexibility. The concept was introduced by David Heinemeier Hansson to describe the philosophy of the Ruby on Rails web framework, but is related to earlier ideas like the concept of "sensible defaults" and the principle of least astonishment in user interface design.

https://en.wikipedia.org/wiki/Convention_over_configuration

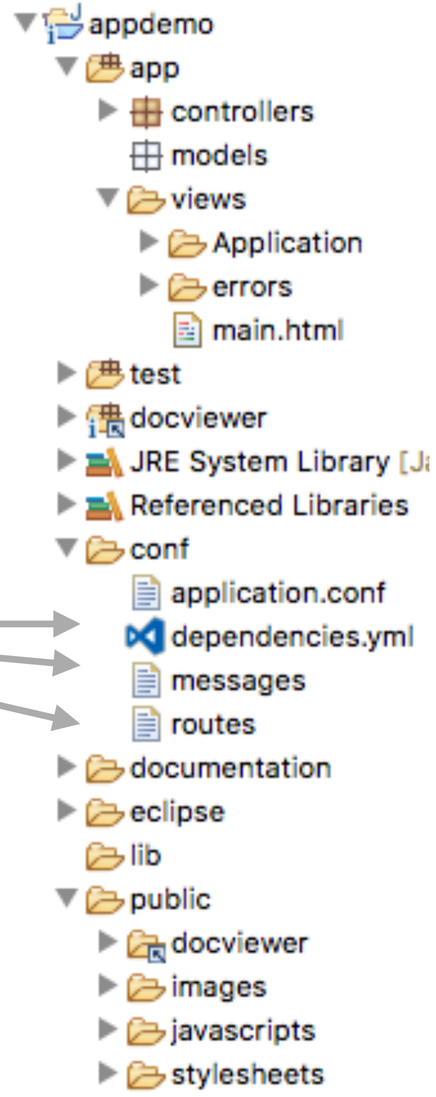
Convention over Configuration in Play 1

```
play new
```

Generates a complete working web app

Considerable range of defaults already configured to ‘just work’

Default can be changed by



Convention over Code - Example

```
const bean = require('jellybean');

bean.setName('Coffee');
bean.setColor('brown');
bean.setSpeckles(false);
```

- Verbose - 3 method calls on the bean object to configure the jellybean.
- Configuration part of the program logic

Convention over Code - Example

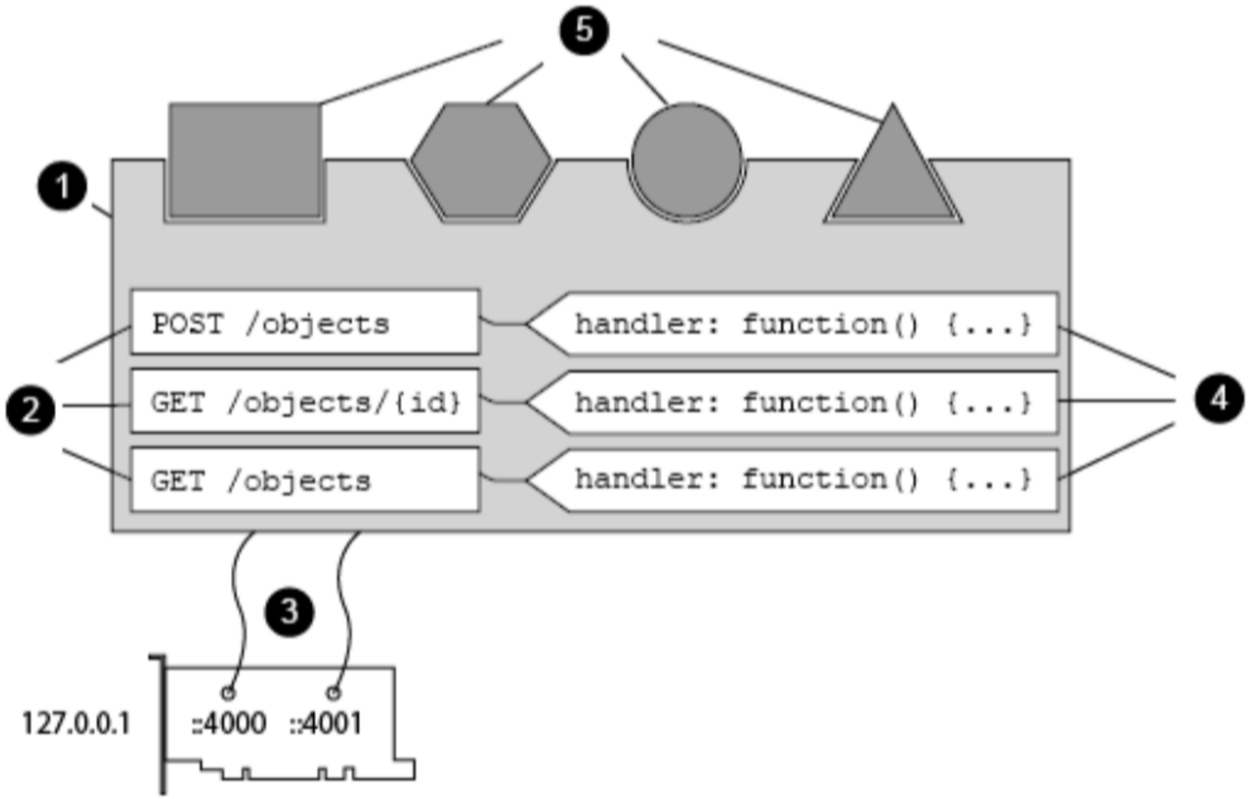
```
const bean = require(‘jellybean’);

const options = {
  name: 'Tutti Frutti',
  color: 'mixed',
  speckles: true
};

bean.config(options);
```

- config method takes options argument
- More flexible because it separates the configuration from the code
- Place all the configurations of jellybeans in a separate file and include them.
- To change the configurations later just update the config.

```
const bean = require('jellybean');

bean.setName('Coffee');
bean.setColor('brown');
bean.setSpeckles(false);
```

```
const bean = require(‘jellybean’);

const options = {
  name: 'Tutti Frutti',
  color: 'mixed',
  speckles: true
};

bean.config(options);
```

Hapi Building Blocks



1. Server
2. Routes
3. Connections
4. Handlers
5. Plugins

Server

index.js

```
'use strict';

const Hapi = require('hapi');

const server = Hapi.server({
  port: 3000,
  host: 'localhost'
});

async function init() {
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
}

process.on('unhandledRejection', err => {
  console.log(err);
  process.exit(1);
});

init();
```

- A server is the container for the hapi application.
- All other Hapi objects are created or used in the context of a server.
- Make connections from your server so the app can speak to the outside world.

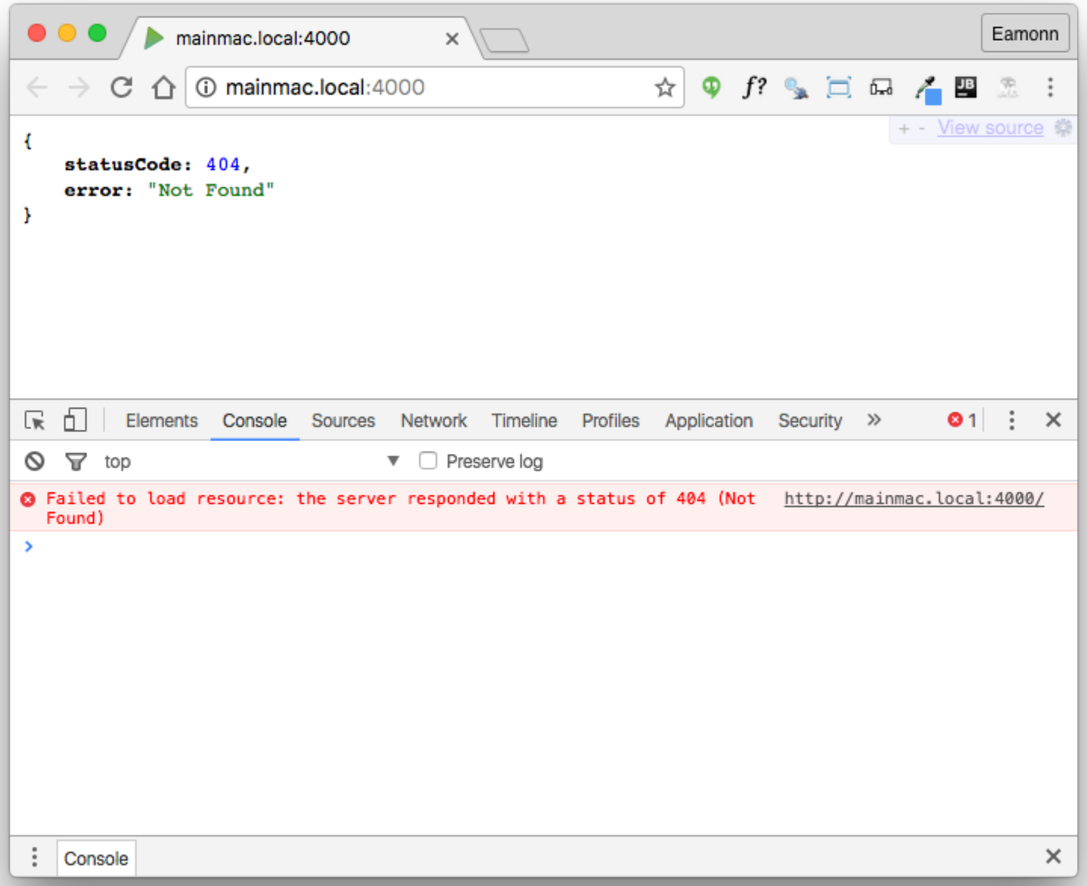
Routes

- Routes in hapi are a way of telling the framework that you're interested in certain types of request.

- Create a route with a set of options, including the HTTP verb (such as GET, POST) and path (for example /about) that you wish to respond to, and add it to a server.

```
const Controller = require('./controller.js');

module.exports = [

  { method: 'GET', path: '/', config: Controller.index },

];
```

No Routes Configured




```
'use strict';

const Hapi = require('hapi');

const server = Hapi.server({
  port: 3000,
  host: 'localhost'
});

async function init() {
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
}

process.on('unhandledRejection', err => {
  console.log(err);
  process.exit(1);
});

init();
```

Configuring Routes

- When a new request arrives at the server, hapi will attempt to find one of the routes that matches the request.

- If it successfully pairs up the request with one of your routes, it will look to your route handler for how to handle the request.

```
const Controller = require('./controller.js');

module.exports = [

  { method: 'GET', path: '/', config: Controller.index },

];
```

```
'use strict';

const Hapi = require('hapi');

const server = Hapi.server({
  port: 3000,
  host: 'localhost'
});

async function init() {
  server.route(require('./routes'));
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
}

process.on('unhandledRejection', err => {
  console.log(err);
  process.exit(1);
});

init();
```

Starting the Server

- server.start called to start the server.
- If there is an error on startup, error details passed in ‘err’ parameter.
- If no error, the server is running, awaiting requests and dispatching to handlers based on the installed routes

```
'use strict';

const Hapi = require('hapi');

const server = Hapi.server({
  port: 3000,
  host: 'localhost'
});

async function init() {
  server.route(require('./routes'));
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
}

process.on('unhandledRejection', err => {
  console.log(err);
  process.exit(1);
});

init();
```

Handlers

- Handlers are the way to tell hapi how it should respond to an HTTP request.
- A handler can take several forms.
- The simplest handler is defined as a JavaScript function with access to a request object and a reply interface.
- The request object provides details about the request.
- Return a value to be rendered by the browser

routes.js

```
const Controller = require('./controller.js');

module.exports = [

  { method: 'GET', path: '/', config: Controller.index },

];
```

controller.js

```
exports.index = {
  handler: function(request, h) {
    return 'Hello!';
  }
};
```

```
exports.index = {
  handler: function(request, h) {
    return 'Hello!';
  }
};
```

The request parameter is an object with details about the end user's request, such as path parameters, an associated payload, authentication information, headers, etc.

h is the response toolkit, an object with several methods used to respond to the request.

1. Servers
2. Connections
3. Routes
4. Handlers

```
exports.index = {
  handler: function(request, reply) {
    return 'Hello!';
  }
};
```

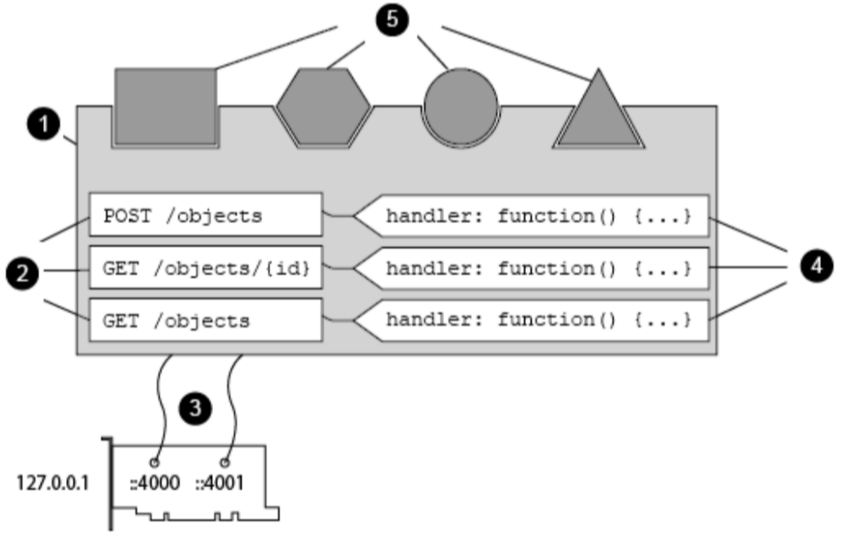
4

```
const Controller = require('./controller.js');

module.exports = [

  { method: 'GET', path: '/', config: Controller.index },

];
```

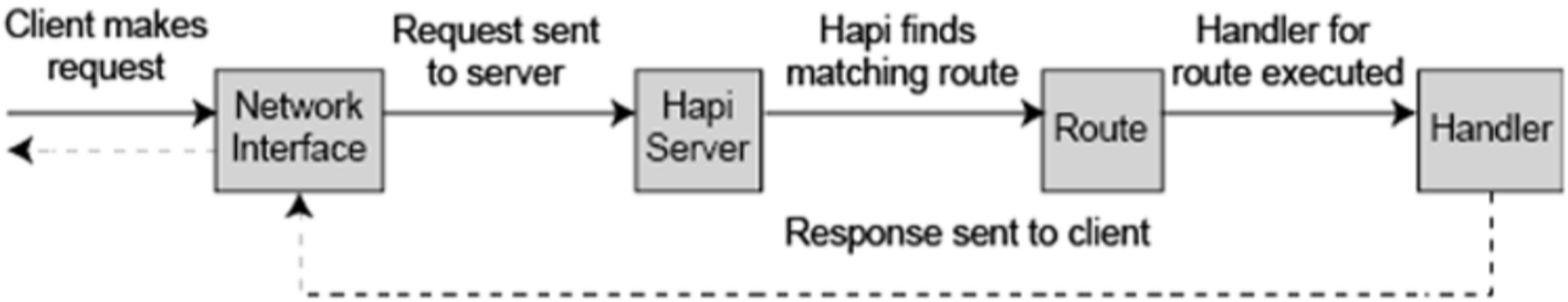
3

```
'use strict';

const Hapi = require('hapi');

const server = Hapi.server({
  port: 3000,
  host: 'localhost'
});

async function init() {
  server.route(require('./routes'));
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
}

process.on('unhandledRejection', err => {
  console.log(err);
  process.exit(1);
});

init();
```

1 2



Hapi Request Handing



Connection -> Server -> Route -> Handler

donation-web [~/repos/modules/wit-hdip-comp-sci/ent-web-prj/donation-web] - .../index.js [donation-web]

donation-web > index.js

Project

- donation-web ~/repos/modules/wit-
  - node_modules library root
  - .gitignore
  - controller.js
  - index.js
  - packag
  - packag
  - routes.
- External L
- Scratches

toolkit.js | controller.js | index.js | routes.js

```
1  'use strict';
2
3  const Hapi = require('hapi');
4
5  const server = Hapi.server( options: {
6    port: 3000,
     : 'localhost'

     .route(require('./routes'));

     function init() {
     t server.start();
     ole.log(`Server running at: ${server.info.uri}`);

     s.on('unhandledRejection', err => {
     ole.log(err);
     ess.exit(1);

     ;
```

- New
- Cut ⌘X
- Copy ⌘C
- Copy Path ⇧⌘C
- Copy Relative Path ⌥⇧⌘C
- Paste ⌘V
- Jump to Source ⌘↓
- Find Usages ⌥F7
- Inspect Code...
- Refactor
- Add to Favorites
- Delete... ⌦
- Mark as Plain Text
- Run 'index.js' ^⇧R
- Debug 'index.js' ^⇧D
- Save 'index.js'
- Reveal in Finder
- Open in Terminal
- Local History
- Git
- Synchronize 'index.js'
- Edit Scopes...
- Compare With... ⌘D
- External Tools
- Diagrams
- Open on GitHub
- Create Gist...
- Reformat with Prettier ⌥⇧⌘P

Run: index

```
/usr/ /modules/wit-hdip-comp-sci/ent-web-prj/donation-web/index.js
Serv
Proc upted by signal 2: SIGINT)
```

4: Run | 6: TODO | 9: Version Control | Terminal | Event Log

12:4 LF UTF-8 2 spaces Git: master

Firefox Developer Edition

http://localhost:4000/

localhost:4000

Hello!

Inspector | Console | Debugger | Style Editor | Performance | Memory | Network

All | HTML | CSS | JS | XHR | Fonts | Images | Media | Flash | WS | Other

One request, 0.01 KB, 0.00 s

Filter URLs

Status	Method	File	Domain
200	GET	/	localhost:400

Headers | Cookies | Params | Response | Timings | Preview

Request URL: `http://localhost:4000/`
Request method: `GET`
Remote address: `127.0.0.1:4000`
Status code: 200 OK
Version: `HTTP/1.1`

Edit and Resend | Raw headers

Request headers:

```
Host: localhost:4000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

Response headers:

```
Cache-Control: no-cache
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Thu, 25 Aug 2016 10:11:52 GMT
Transfer-Encoding: chunked
Vary: accept-encoding
```

Plugins

- Plugins are a way of extending servers with new functionality.
- Plugins can extend a server with some global utility such as logging all requests or adding caching to responses.
- There are many existing plugins available as npm packages that deal with things like authentication and logging, written by the hapi core team and community.
- It’s also possible to create your own plugins that divide your application into smaller logical chunks that are easier to maintain or even replace or remove altogether at a later date.

Plugins Example

https://github.com/hapijs/inert

inert

Static file and directory handlers plugin for hapi.js.

build passing

Lead Maintainer - Gil Pedersen

inert provides new handler methods for serving static files and directories, as well as adding a `h.file()` method to the toolkit, which can respond with file based resources.

Features

- Files are served with cache friendly `last-modified` and `etag` headers.
- Generated file listings and custom indexes.
- Precompressed file support for `content-encoding: gzip` responses.
- File attachment support using `content-disposition` header.

Index

Plugin Configuration & Registration

- Plugins often take their configuration as an object, specifying various feature initial values (not in this case though).

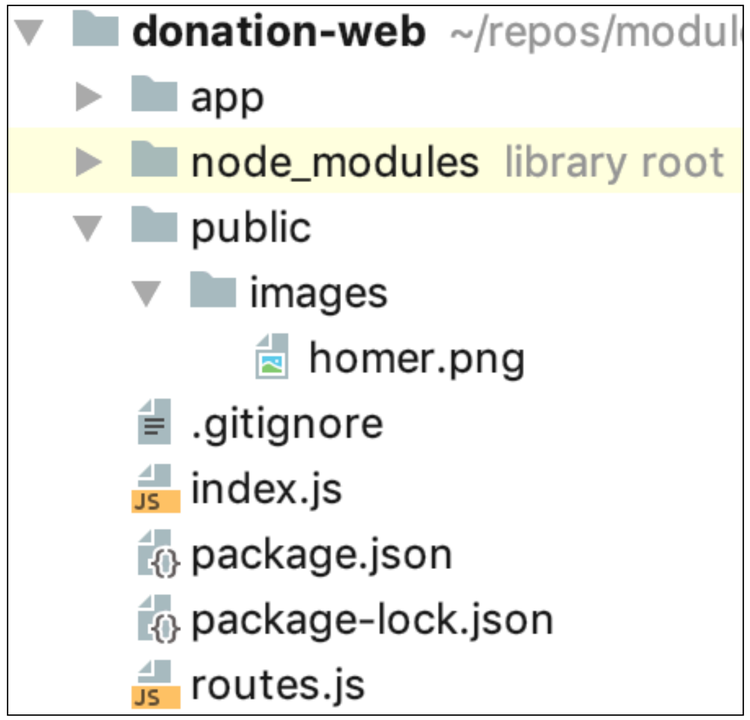
- Plugins are then registered - and only when this is complete is the service started

```
...
async function init() {
  await server.register(require('inert'));
  server.route(require('./routes'));
  await server.start();
  console.log(`Server running at: ${server.info.uri}`);
}
...
```

inert in action

Permits this type of route

```
{
  method: 'GET',
  path: '/{param*}',
  handler: {
    directory: {
      path: './public'
    }
  }
}
```



Many other options & patterns possible