



Core Concepts & Setup

Server Initialization

Creating a basic Hapi.js server instance.

```
const Hapi = require('@hapi/hapi');

const start = async function() {

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

  await server.start();

  console.log(`Server started at: ${server.info.uri}`);
}

start();
```

Explanation:

- `require('@hapi/hapi')`: Imports the Hapi.js library.
- `Hapi.server({...})` : Creates a new server instance with configuration options.
- `port` : Specifies the port number for the server.
- `host` : Specifies the hostname or IP address.
- `server.start()` : Starts the server, listening for incoming requests.

Basic Routing

Defining a simple route to handle GET requests.

```
server.route({
  method: 'GET',
  path: '/',
  handler: (request, h) => {
    return 'Hello, world!';
  }
});
```

Explanation:

- `method` : Specifies the HTTP method for the route (e.g., 'GET', 'POST').
- `path` : Defines the URL path for the route.
- `handler` : A function that handles incoming requests to the route.
- `request` : An object containing information about the incoming request.
- `h` : The response toolkit object, used to build and send responses.

Route Configuration Options

<code>method</code>	HTTP method (GET, POST, PUT, DELETE, etc.).
<code>path</code>	URL path for the route.
<code>handler</code>	Function to process the request and return a response.
<code>config</code>	Object for route-specific configurations (e.g., validation, authentication).

Request Handling & Validation

Accessing Request Data

Accessing data from various parts of the request object.

```
handler: (request, h) => {
  const query = request.query; // Query parameters
  const params = request.params; // Route parameters
  const payload = request.payload; // Request body

  console.log('Query:', query);
  console.log('Params:', params);
  console.log('Payload:', payload);

  return 'Data received!';
}
```

Payload Validation with Joi

Using Joi for request payload validation.

```
const Joi = require('joi');

server.route({
  method: 'POST',
  path: '/user',
  handler: (request, h) => {
    return 'User created!';
  },
  options: {
    validate: {
      payload: Joi.object({
        username: Joi.string().alphanumeric().min(3).max(30).required(),
        password: Joi.string().pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')).required(),
        email: Joi.string().email().required()
      })
    }
  }
});
```

Query Parameter Validation

query:	Validates query parameters. Joi.object({ ...})
params:	Validates route parameters. Joi.object({ ...})
failAction:	Defines what happens when validation fails (e.g., 'log', 'error'). '...'

Plugins & Decorators

Registering Plugins

Registering a plugin to extend server functionality.

```
const myPlugin = {
  name: 'my-plugin',
  version: '1.0.0',
  register: async function (server,
options) {
  server.route({
    method: 'GET',
    path: '/plugin',
    handler: (request, h) => {
      return 'Plugin route!';
    }
  });
}
};
```

```
const start = async function() {

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

  await server.register({
    plugin: myPlugin,
    options: { /* plugin options */ }
  });

  await server.start();

  console.log(`Server started at:
${server.info.uri}`);
}

start();
```

Server Decorators

Adding custom methods to the server object.

```
server.decorate('toolkit', 'myHelper',
function (param) {
  return `Helper called with
${param}`;
});

server.route({
  method: 'GET',
  path: '/helper',
  handler: (request, h) => {
    return
server.toolkit.myHelper('value');
  }
});
```

Request Decorators

server.decorate('request', ...)	Adds a method to the request object.
server.decorate('toolkit', ...)	Adds a method to the response toolkit.
server.decorate('server', ...)	Adds a method to the server object.

Advanced Features

Authentication Strategies

Implementing authentication strategies using plugins like `hapi-auth-jwt2` or `hapi-auth-basic`.

```
await server.register(require('hapi-auth-jwt2'));

server.auth.strategy('jwt', 'jwt',
{
  key: 'YourSecretKey',
  validate: async (decoded, request)
=> {

  // Perform validation logic here
  const isValid = true; // Replace
with your validation check

  return {
    isValid: isValid
  };
},
verifyOptions: { algorithms: [
'HS256' ] }
});

server.auth.default('jwt'); // Set
default authentication strategy
```

Caching Strategies

Configuring caching strategies to improve performance.

```
const CatboxMemory =
require('@hapi/catbox-memory');

const start = async function() {

  const server = Hapi.server({
    port: 3000,
    host: 'localhost',
    cache: [
      {
        provider: {
          constructor:
CatboxMemory
        }
      }
    ]
  });

  server.route({
    method: 'GET',
    path: '/cache',
    handler: async (request, h) => {

      const item = await
server.cache.get({ key: 'myKey' });

      return item;
    },
    options: {
      cache: {
        expiresIn: 60000, // 1
minute
        privacy: 'private'
      }
    }
  });

  await server.start();
}

start();
```

Middleware and Lifecycle Methods

<code>onPreHandler</code>	Runs before the handler function.
<code>onPostHandler</code>	Runs after the handler function, before sending the response.
<code>onPreResponse</code>	Runs before sending the response, allowing modification.