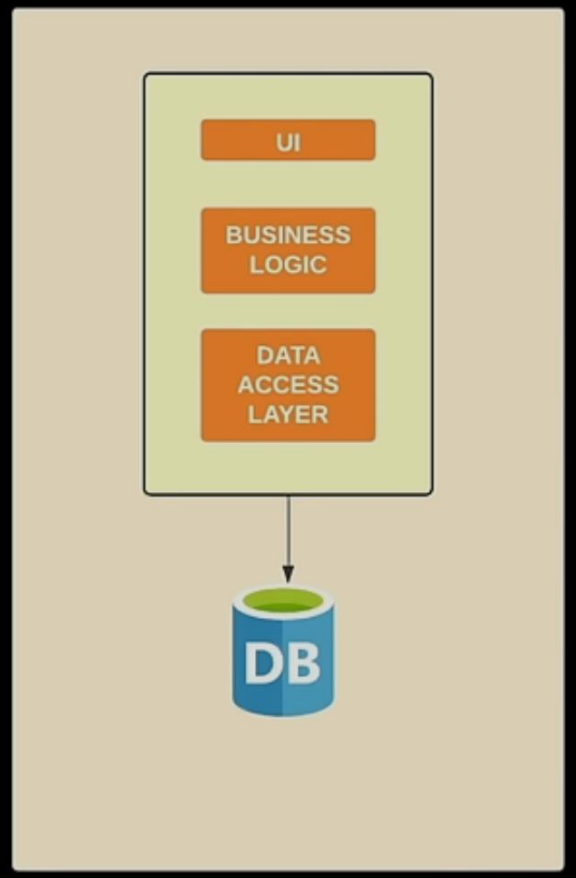
Moleculer Framework

ANJU MUNOTH

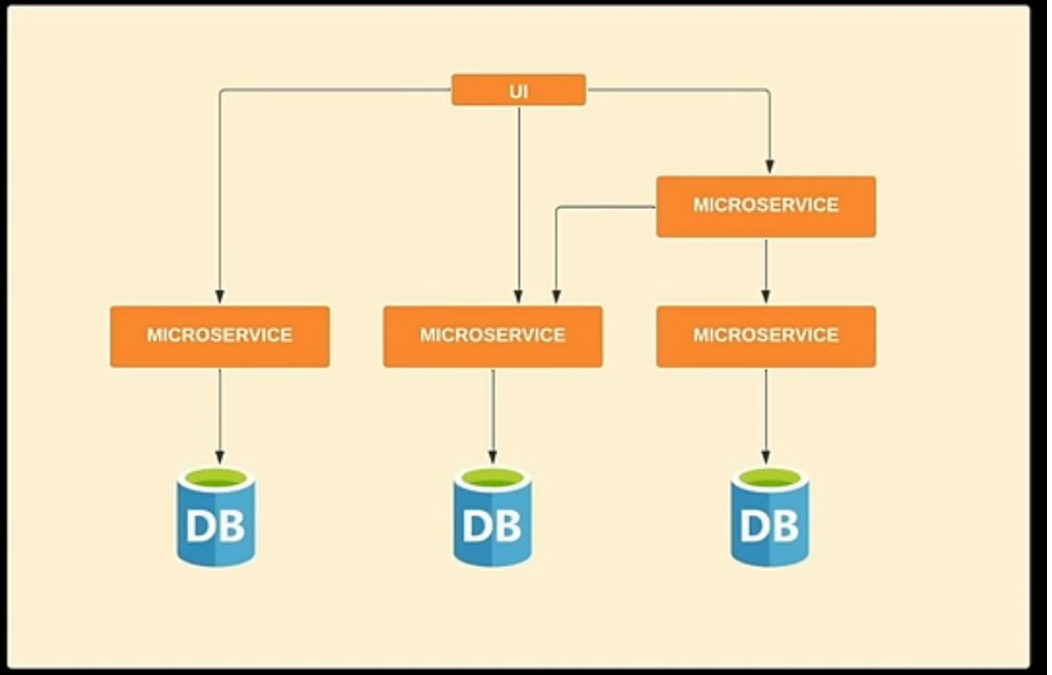
Monolithic Architecture



- Built and deployed as a single unit
- Easy to develop, test & deploy
- Cheaper & simpler infrastructure
- Usually built with single tech stack
- Can run into issues when scaling

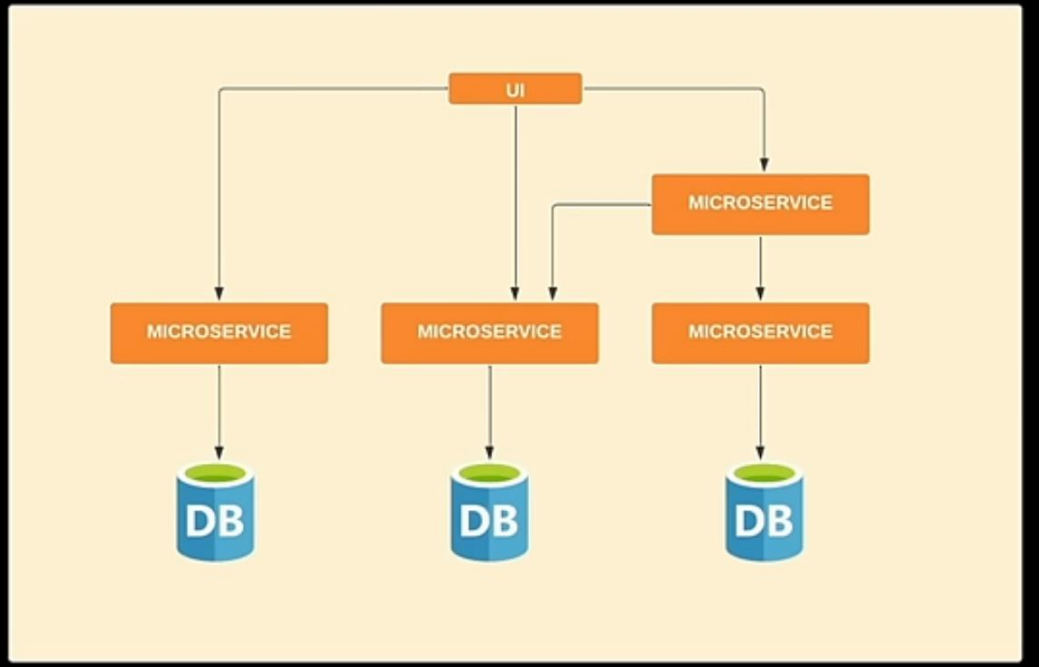
Microservices Architecture

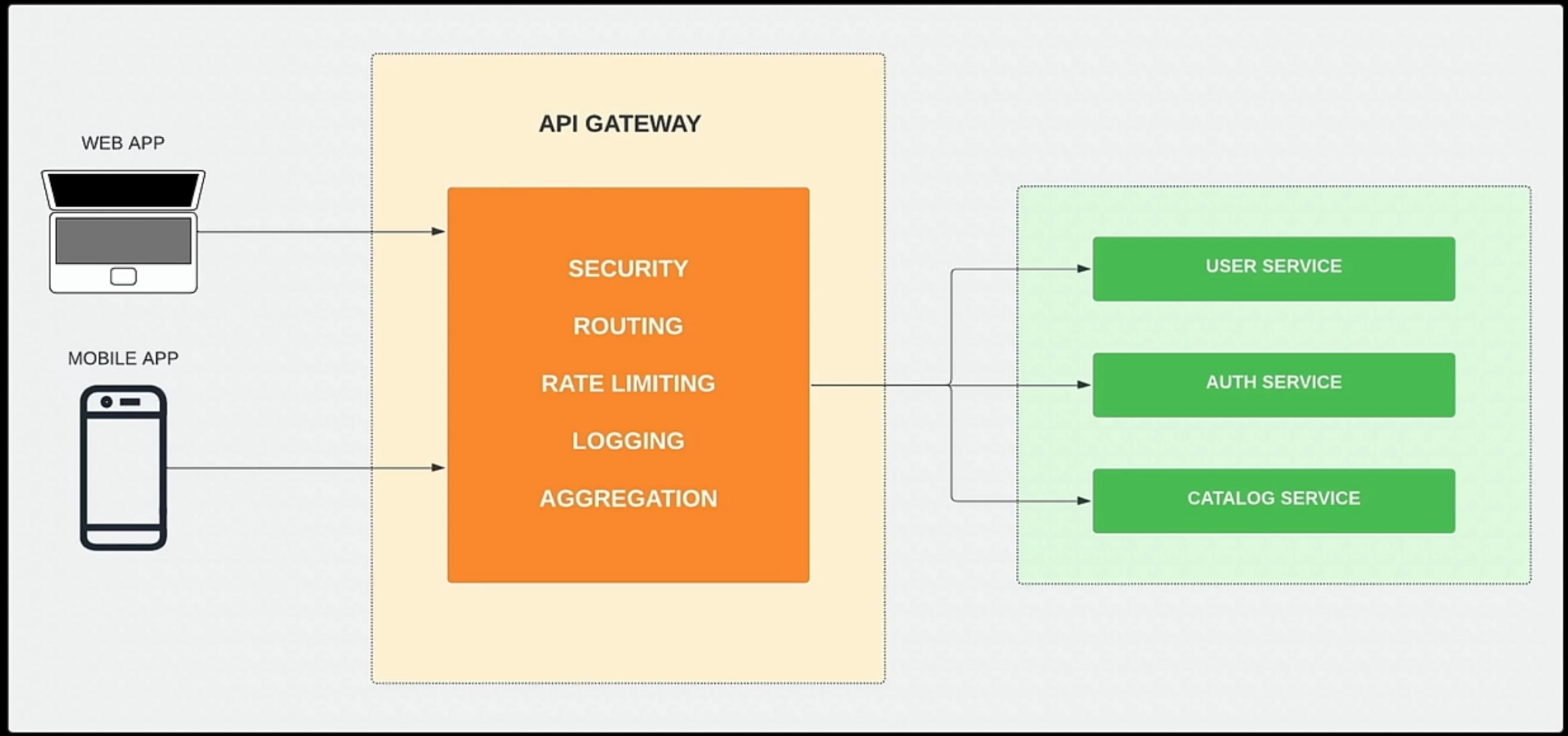
- Collection of small, loosely coupled services
- Self-contained, can be developed, deployed and scaled independently
- Communicate with each other with protocols like HTTP



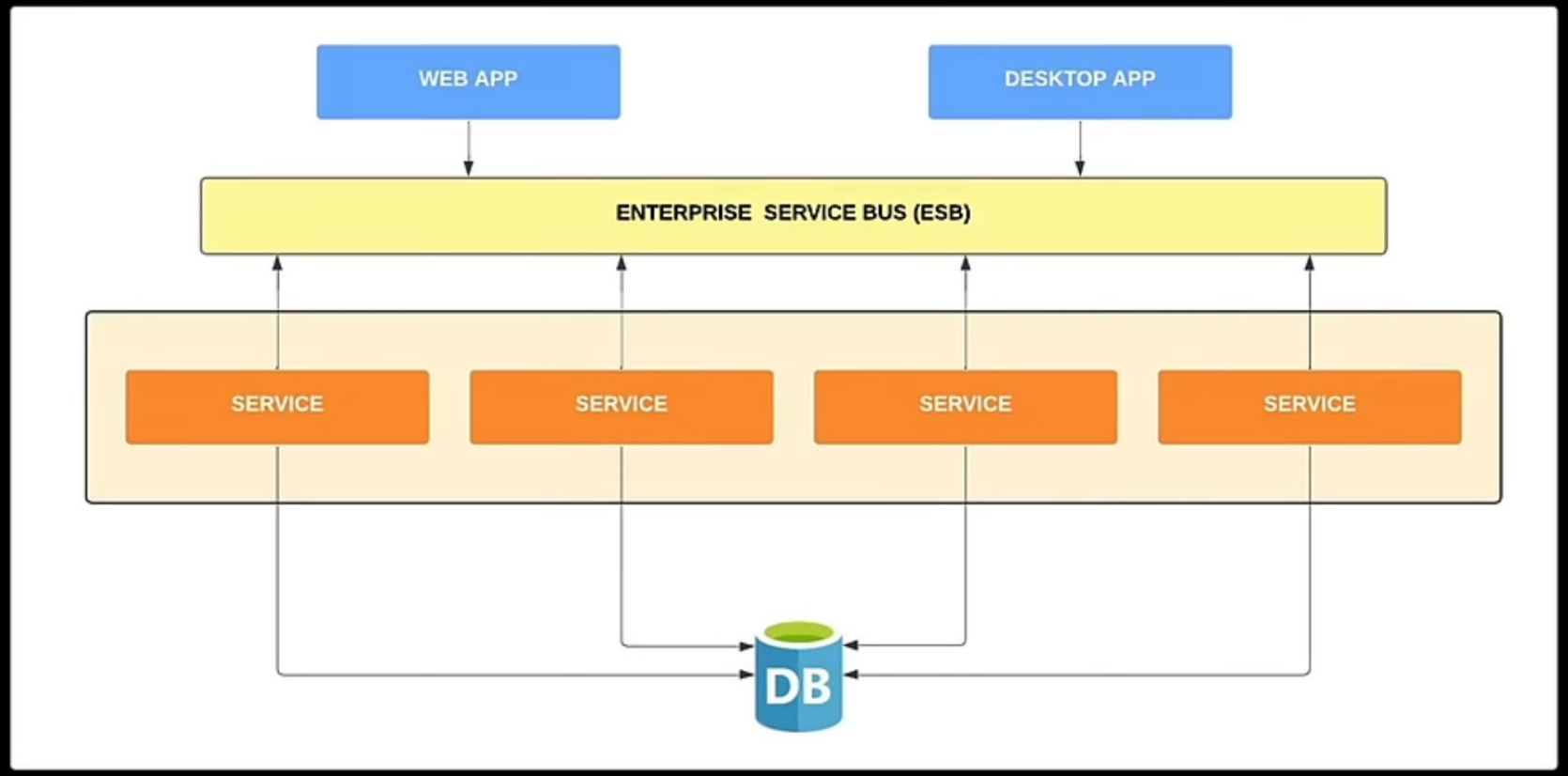
Microservices Architecture

- Not tied to any specific technology, language, framework, etc
- Adds more complexity to an application
- Larger infrastructure and more expensive than monolith




WEB APP
MOBILE APP
API GATEWAY
SECURITY
ROUTING
RATE LIMITING
LOGGING
AGGREGATION
USER SERVICE
AUTH SERVICE
CATALOG SERVICE

Service Oriented Architecture (SOA)



Differences From Microservices:

- Communication
- Middleware Complexity
- Granularity
- Data Management

Pros & Cons

- Scalability
- Flexibility
- Resilience
- Modular & Decentralized
- Team Autonomy

- Complexity
- Operational Overhead
- Data Management
- Development Time
- Debugging

Moleculer Framework

- Fast, modern and powerful microservices framework for Node.js.
- Helps to build efficient, reliable & scalable services.
- Moleculer provides many features for building and managing your microservices.

- Promise-based solution (async/await compatible)
- request-reply concept
- support event driven architecture with balancing
- built-in service registry & dynamic service discovery
- load balanced requests & events (round-robin, random, cpu-usage, latency, sharding)
- many fault tolerance features (Circuit Breaker, Bulkhead, Retry, Timeout, Fallback)
- plugin/middleware system
- support versioned services
- support Streams
- service mixins
- built-in caching solution (Memory, MemoryLRU, Redis)
- pluggable loggers (Console, File, Pino, Bunyan, Winston, Debug, Datadog, Log4js)
- pluggable transporters (TCP, NATS, MQTT, Redis, NATS Streaming, Kafka, AMQP 0.9, AMQP 1.0)
- pluggable serializers (JSON, Avro, MsgPack, Protocol Buffer, Thrift)
- pluggable parameter validator
- multiple services on a node/server
- master-less architecture, all nodes are equal
- parameter validation with fastest-validator
- built-in metrics feature with reporters (Console, CSV, Datadog, Event, Prometheus, StatsD)
- built-in tracing feature with exporters (Console, Datadog, Event, Jaeger, Zipkin)
- official API gateway, Database access and many other modules…

Features of moleculer framework

- **Microservices Architecture:** Moleculer is designed to facilitate the development of microservices-based applications.
  - Microservices architecture is an approach to software development where an application is composed of loosely coupled, independently deployable services.
- **Services Oriented:** In Moleculer, each microservice is a collection of independent components called services.
  - Services can communicate with each other using built-in mechanisms such as request-response or publish-subscribe patterns.
- **Service Communication**: Moleculer provides a built-in communication layer that allows services to communicate with each other.
  - Supports various transporters such as TCP, Redis, MQTT, and more.
  - Framework abstracts away the complexities of inter-service communication,making it easier to establish communication channels.

Features of moleculer framework

- **Load Balancing and Fault Tolerance:** Moleculer includes built-in support for load balancing and fault tolerance.
  - Can distribute requests among multiple instances of a service to ensure high availability and reliability.
  - Ensures even utilization of resources and helps prevent bottlenecks.
- **Service Discovery:** Service discovery is a critical aspect of microservices architectures, and Moleculer provides mechanisms for services to discover and communicate with each other dynamically, allowing services to dynamically locate and communicate with each other.
- **Scalability**: Moleculer is designed to be scalable. It allows you to scale your microservices horizontally by running multiple instances of the same service across different nodes to handle increased loads.

Features of moleculer framework

- **Transporter Abstraction:** Moleculer abstracts away the underlying transport layer, allowing you to choose different transporters based on your needs.
  - Makes it easy to switch between different transport mechanisms without affecting your application logic.
- **Fault Tolerance:** Microservices applications need to be resilient to failures.
  - Provides features such as automatic retries and circuit breakers to enhance fault tolerance in a microservices environment and ensure that applications remain operational even in the presence of failures.
- **Built-in middleware:** Moleculer supports middleware, which allows you to add cross-cutting concerns such as authentication, logging, and caching to your services easily.
- **Community and Ecosystem:** Moleculer has an active community and ecosystem that contribute to its development and provide extensions, plugins, and integrations.
  - Community support can be valuable for developers seeking assistance and collaboration.

Features of moleculer framework

- **Extensibility**: Moleculer is extensible and allows you to integrate with other libraries and tools.
  - Has a pluggable architecture that enables you to add custom functionality to your microservices.
- **Monitoring and Tracing:** Moleculer provides built-in support for monitoring and tracing your microservices.
  - Can integrate it with tools like Prometheus and Jaeger for enhanced observability, providing insights into the performance and behavior of services.
- **Flexibility**: Moleculer allows developers to design and implement services independently.
  - Crucial in dynamic environments where different services might have different release cycles, dependencies, and scaling requirements.
- **Development Productivity:** Moleculer includes features that enhance development productivity, such as service introspection, automatic generation of documentation, and a pluggable architecture that allows for easy integration with other tools and libraries.

Core components of Moleculer Framework

- **Service**: A service in Moleculer is an independent and modular component that performs a specific function.
  - Services can communicate with each other and expose actions, events, and methods.
- **Action**: Actions are methods or functions exposed by a service that can be invoked by other services or clients.
  - Actions are the primary way for services to interact with each other.
- **Event**: Events are messages that services can emit and listen to.
  - Allow for asynchronous communication between services.
  - When an event is emitted, other services that subscribe to that event can react accordingly.

Moleculer Framework

- **Broker**: Broker is the core component of Moleculer responsible for managing services, handling communication between them, and providing various features such as load balancing, service discovery, and fault tolerance.
- **Transporter**: Transporter is a communication layer that allows services to communicate with each other.
  - Moleculer supports various transporters, including TCP, Redis, MQTT, and more.
- **Node**: A node in Moleculer represents an instance of the application running on a specific machine or container.
  - Multiple nodes can form a Moleculer cluster.
- **Cluster**: Consists of multiple nodes working together to provide a distributed and scalable environment.
  - Enables load balancing, fault tolerance, and the distribution of services across multiple instances.

Moleculer Framework

- **Middleware**: Middleware functions in Moleculer are used to perform tasks such as logging, authentication, and validation.
  - Middleware can be applied to actions to add cross-cutting concerns to the service.
- **Service Registry:** Service discovery is the process by which services locate and communicate with each other.
  - Moleculer provides mechanisms for service discovery, allowing services to find and interact with each other dynamically.
- **Service Discovery:** Service discovery is the process by which services locate and communicate with each other.
  - Moleculer provides mechanisms for service discovery, allowing services to find and interact with each other dynamically.

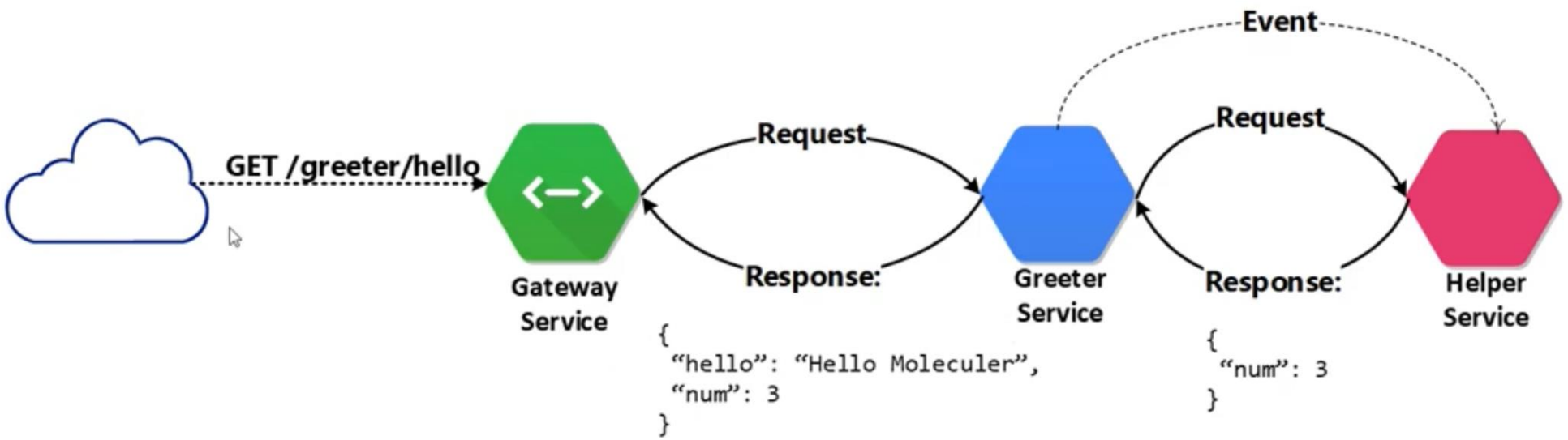
Moleculer Framework

- **Load Balancing:** Moleculer includes built-in load balancing mechanisms that distribute requests among multiple instances of a service.
  - Ensures even utilization of resources and prevents overloading of specific services.
- **Circuit Breaker:** Circuit breaker pattern is used for fault tolerance in Moleculer.
  - Prevents the continuous execution of actions that are likely to fail, helping to isolate failing services and improve overall system resilience.
- **Gateway:** A gateway in Moleculer is a service that provides a single entry point for client applications to interact with various microservices.
  - Can handle API composition and routing.
  - Gateway is a regular Moleculer service running a (HTTP, WebSockets, etc.) server. It handles the incoming requests, maps them into service calls, and then returns appropriate responses.

Why Use It?

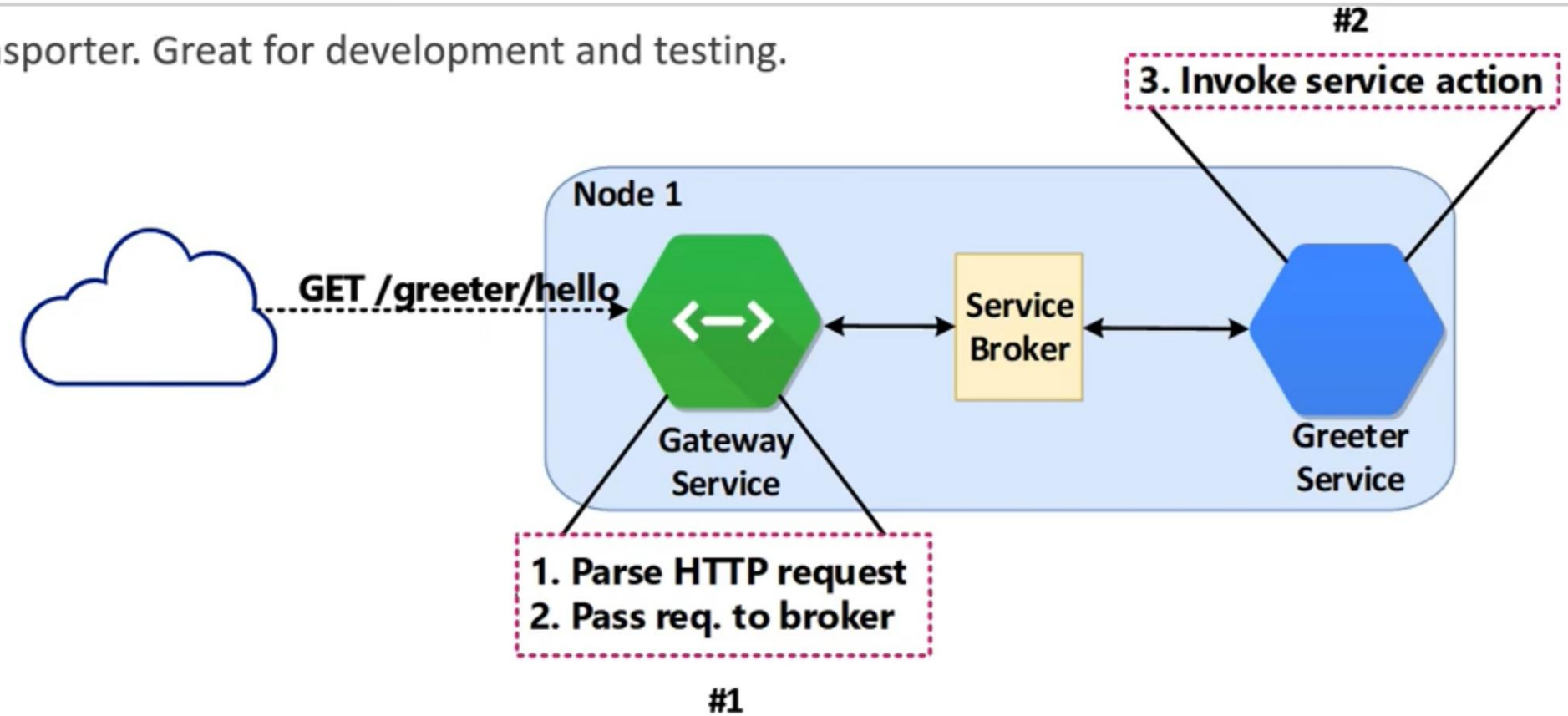
- Fast & Efficient
- Scalable & Progressive
- Supports
  - Request/Reply pattern
  - Events
- Transport Independent
  - Transporters (NATS, MQTT, TCP, Redis, NATS Streaming, Kafka)
- Built-in
  - Load balancing mechanisms (Round-Robin, Random, CPU-usage, Latency)
  - Fault tolerance mechanisms (Circuit Breaker, Retry, Timeout, Fallback, Bulkhead)
  - Validators (Fastest-validator, JSON Schema, Joi)
  - Cache (Memory, Redis, LRU)
  - Health monitoring & metrics
  - Dynamic service discovery & registry

Demo Project



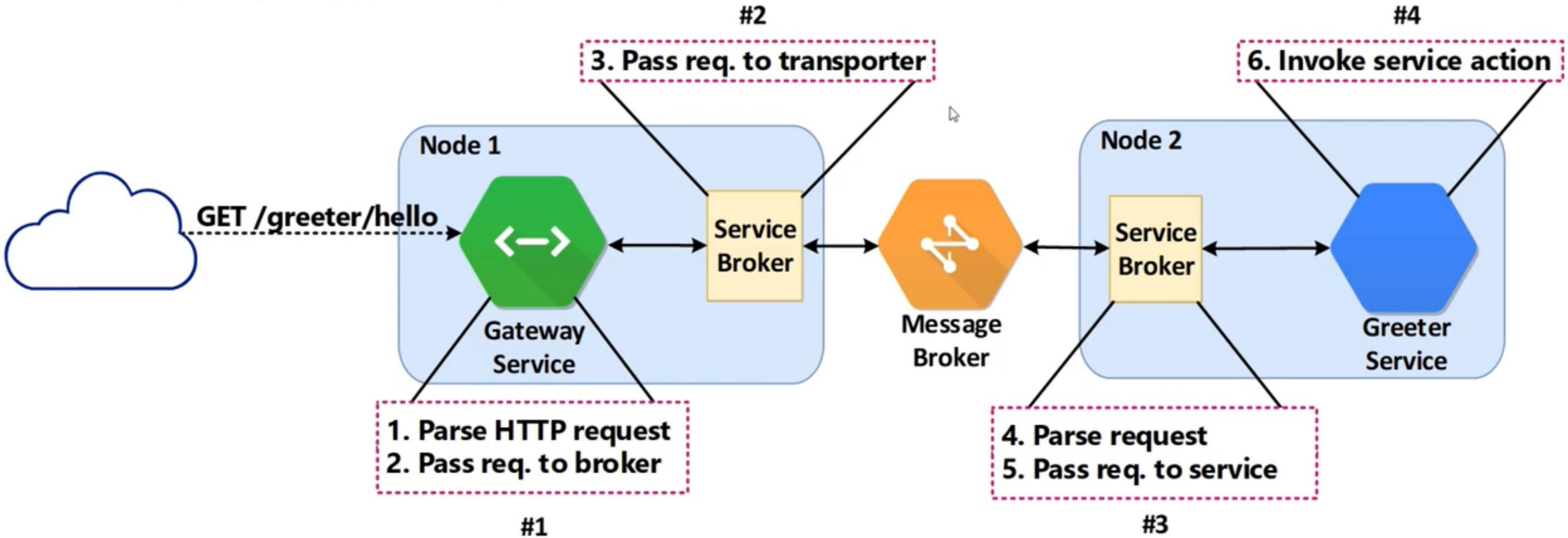
Monolith Deployment

No transporter. Great for development and testing.



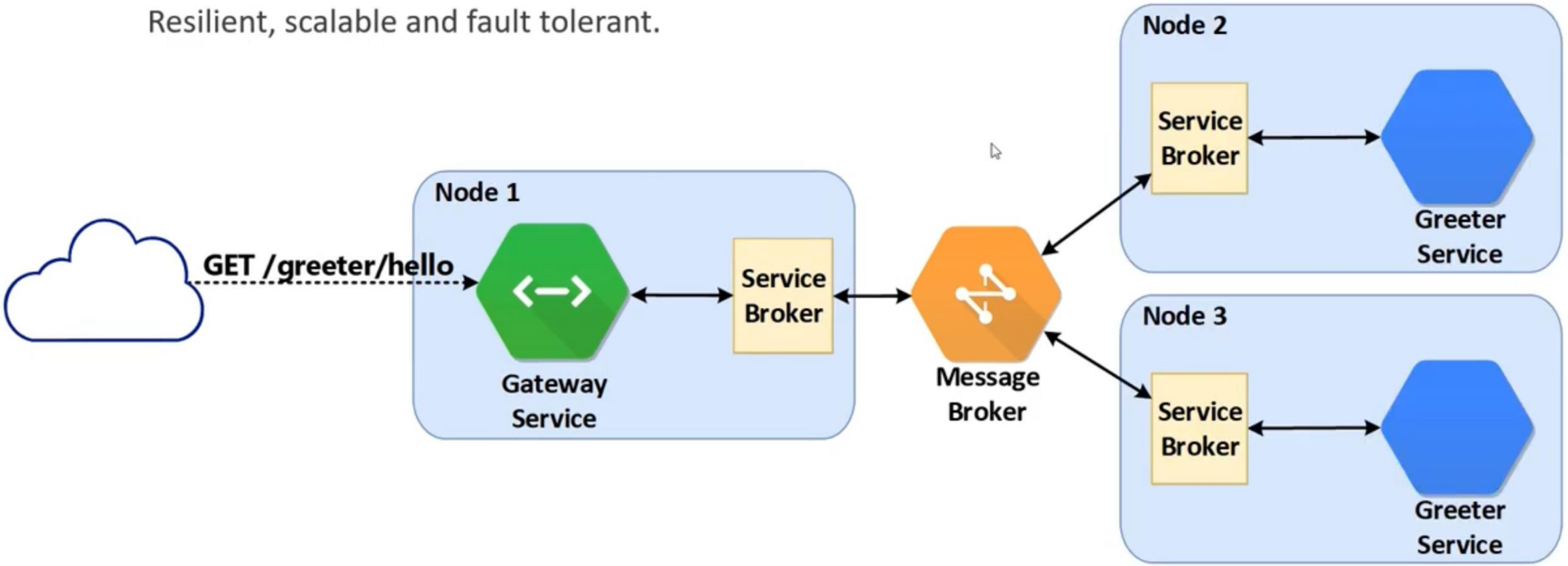
Distributed Deployment

Resilient, scalable and fault tolerant.



Distributed Deployment

Resilient, scalable and fault tolerant.



Core Concepts

Service

- It's simple a JavaScript module
- Contains a set of functions

Node

- An OS process
- Hosts one or many **Services**

ServiceBroker

- **Service**(s) management
- Communication between **Services**
- One per **Node**

Quick setup of project

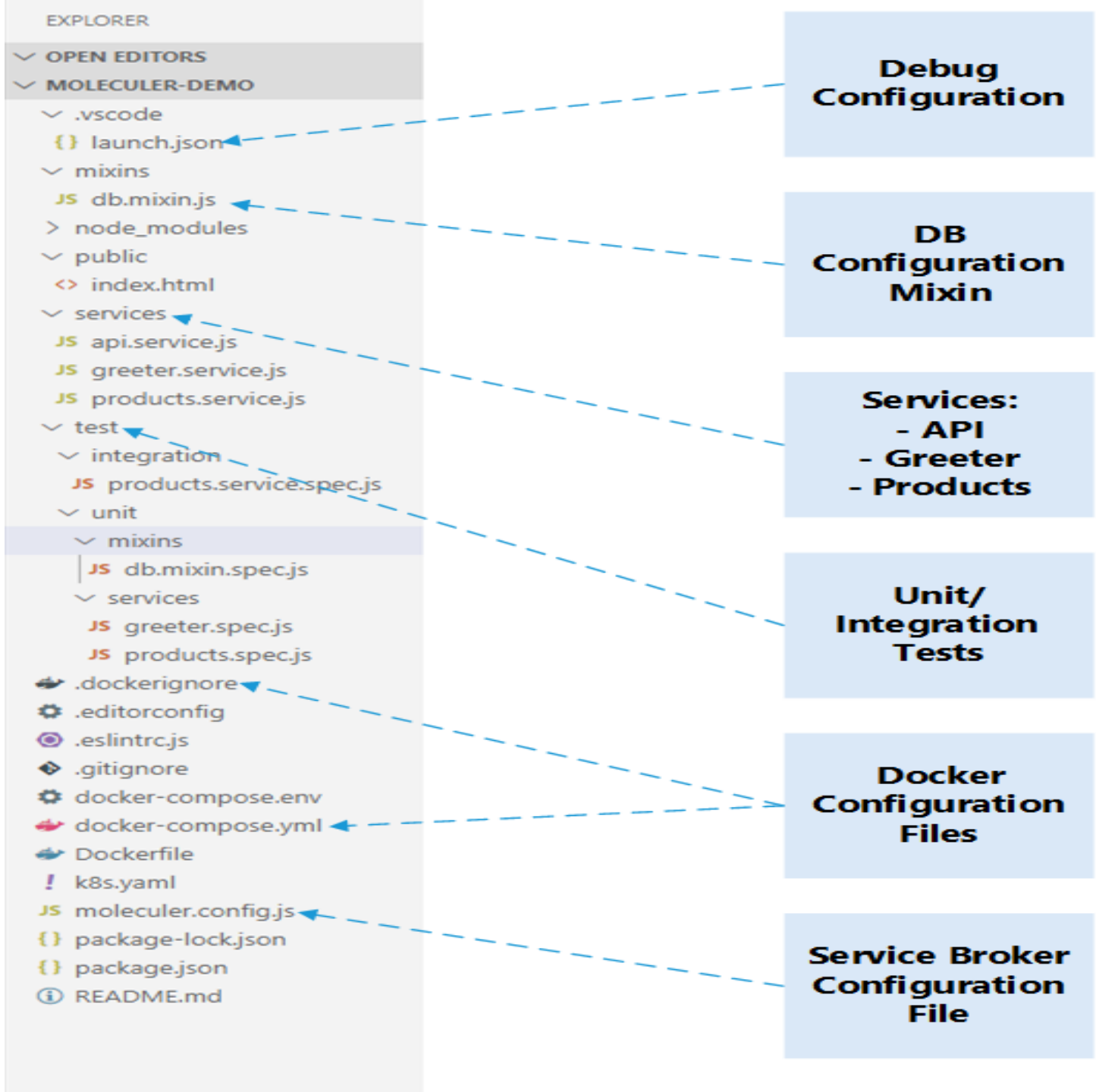
- `npm i moleculer-cli –g`

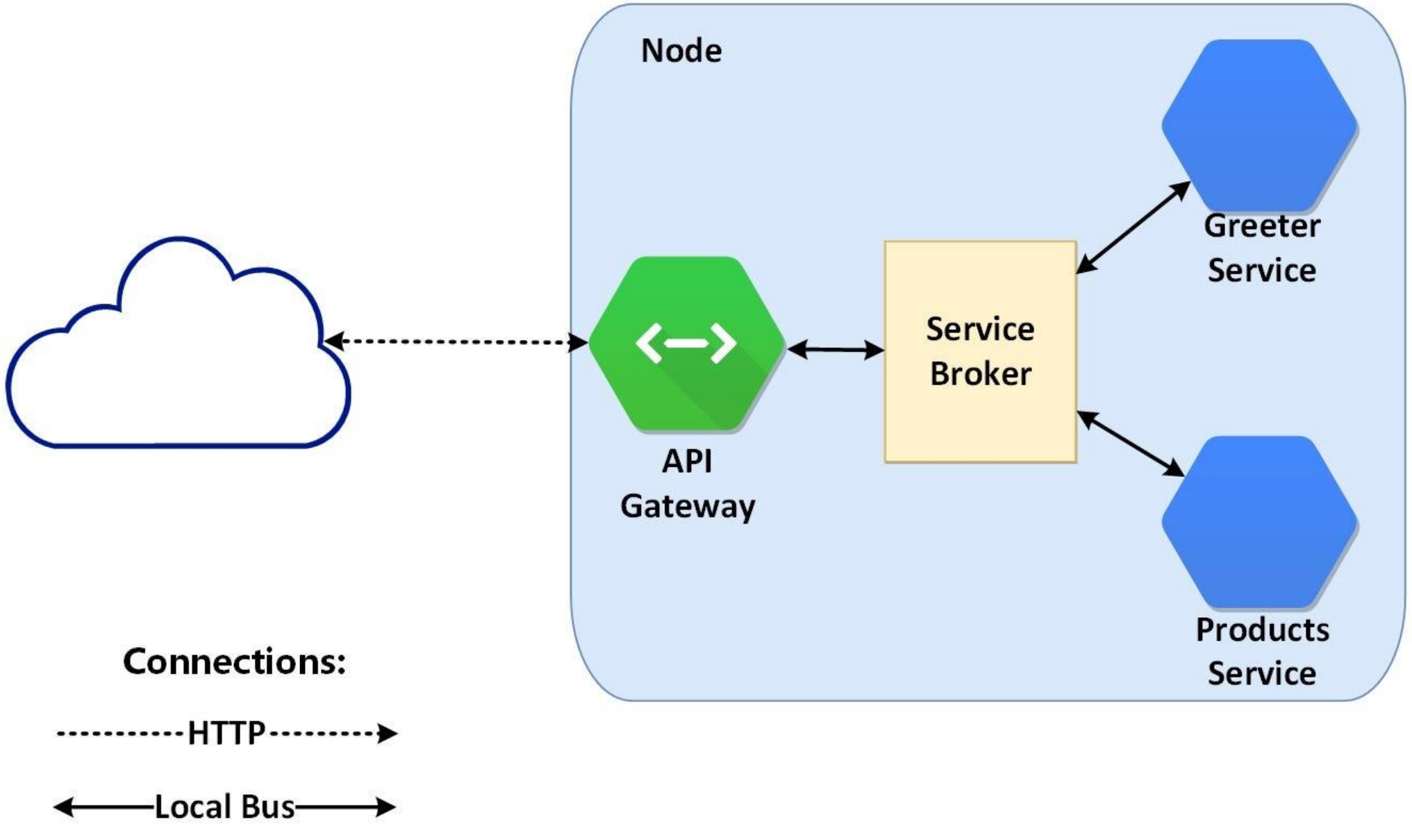
- `moleculer init project moleculer-demo`
- After downloading the template, the CLI is going to ask you some questions. Press “*Enter”* for every prompt. This will configure the template and install the dependencies.
- `npm run dev`

Project setup

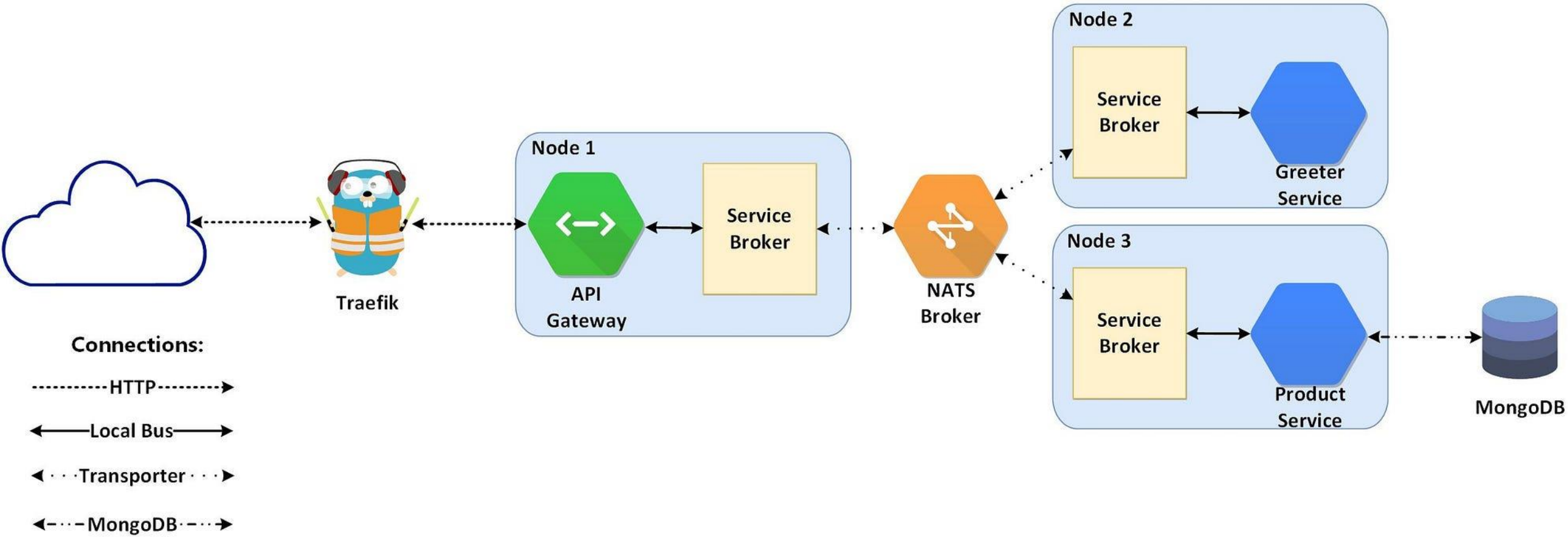
- Inside of the project you will find:
- Three Moleculer (micro)services:

- ***api.service.js*** — Service responsible for handling the incoming HTTP requests.
- ***greeter.service.js*** — Simple “Hello World” service with two service actions. One returns “Hello Moleculer” and the other one returns “Welcome <name>”, where the <name> is passed as an input parameter.
- ***products.service.js*** — DB service with CRUD methods that uses NeDB during the development and MongoDB in production.

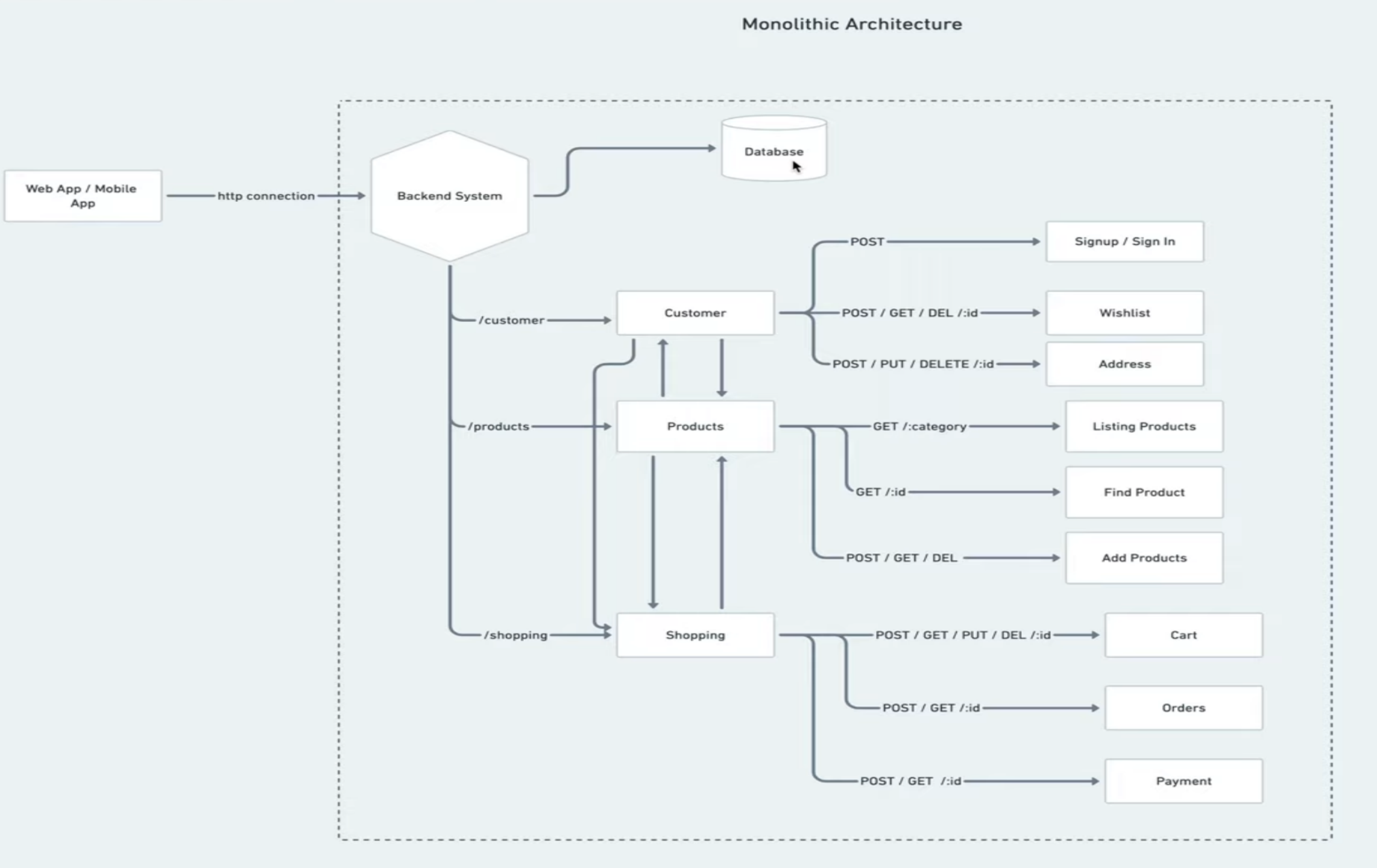

EXPLORER
OPEN EDITORS
MOLECULER-DEMO
.vscode
launch.json
mixins
db.mixin.js
node_modules
public
index.html
services
api.service.js
greeter.service.js
products.service.js
test
integration
products.service.spec.js
unit
mixins
db.mixin.spec.js
services
greeter.spec.js
products.spec.js
.dockerignore
.editorconfig
.eslintrc.js
.gitignore
docker-compose.env
docker-compose.yml
Dockerfile
k8s.yaml
moleculer.config.js
package-lock.json
package.json
README.md
Debug Configuration
DB Configuration Mixin
Services:
- API
- Greeter
- Products
Unit/ Integration Tests
Docker Configuration Files
Service Broker Configuration File



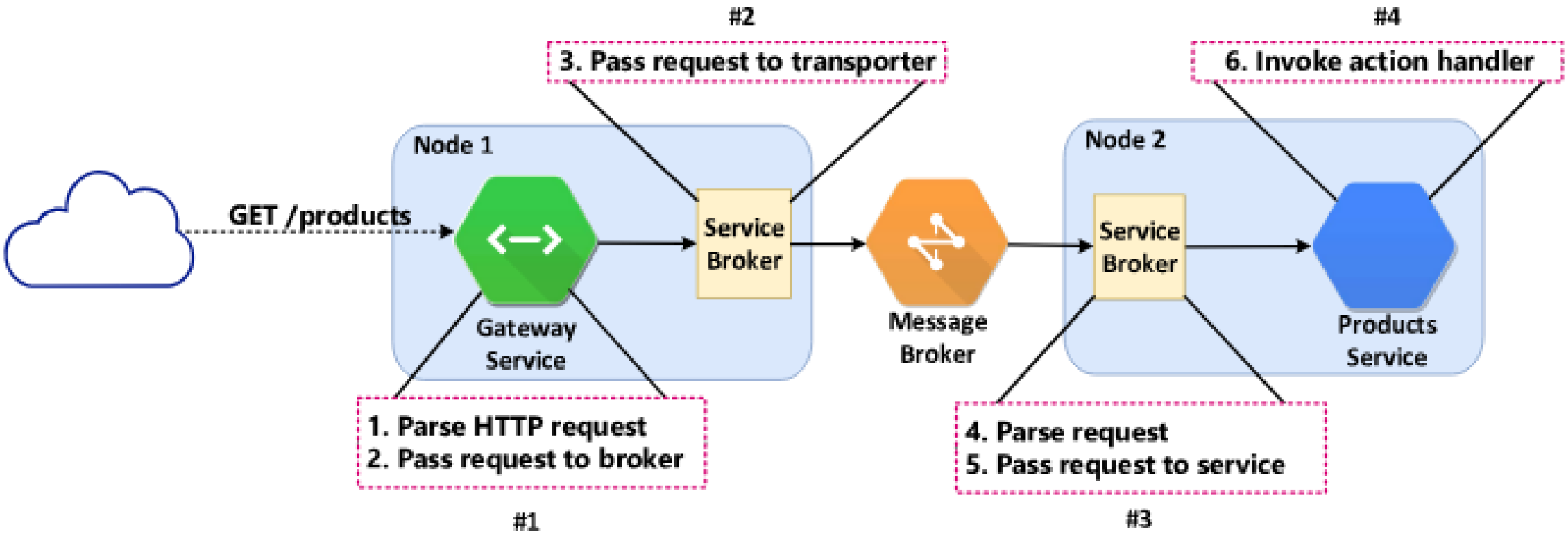
Development configuration



Production configuration


Monolithic Architecture
Web App / Mobile App
http connection
Backend System
Database
/customer
Customer
POST
Signup / Sign In
POST / GET / DEL /:id
Wishlist
POST / PUT / DELETE /:id
Address
/products
Products
GET /:category
Listing Products
GET /:id
Find Product
POST / GET / DEL
Add Products
/shopping
Shopping
POST / GET / PUT / DEL /:id
Cart
POST / GET /:id
Orders
POST / GET /:id
Payment

Moleculer Framework Architecture



Architecture

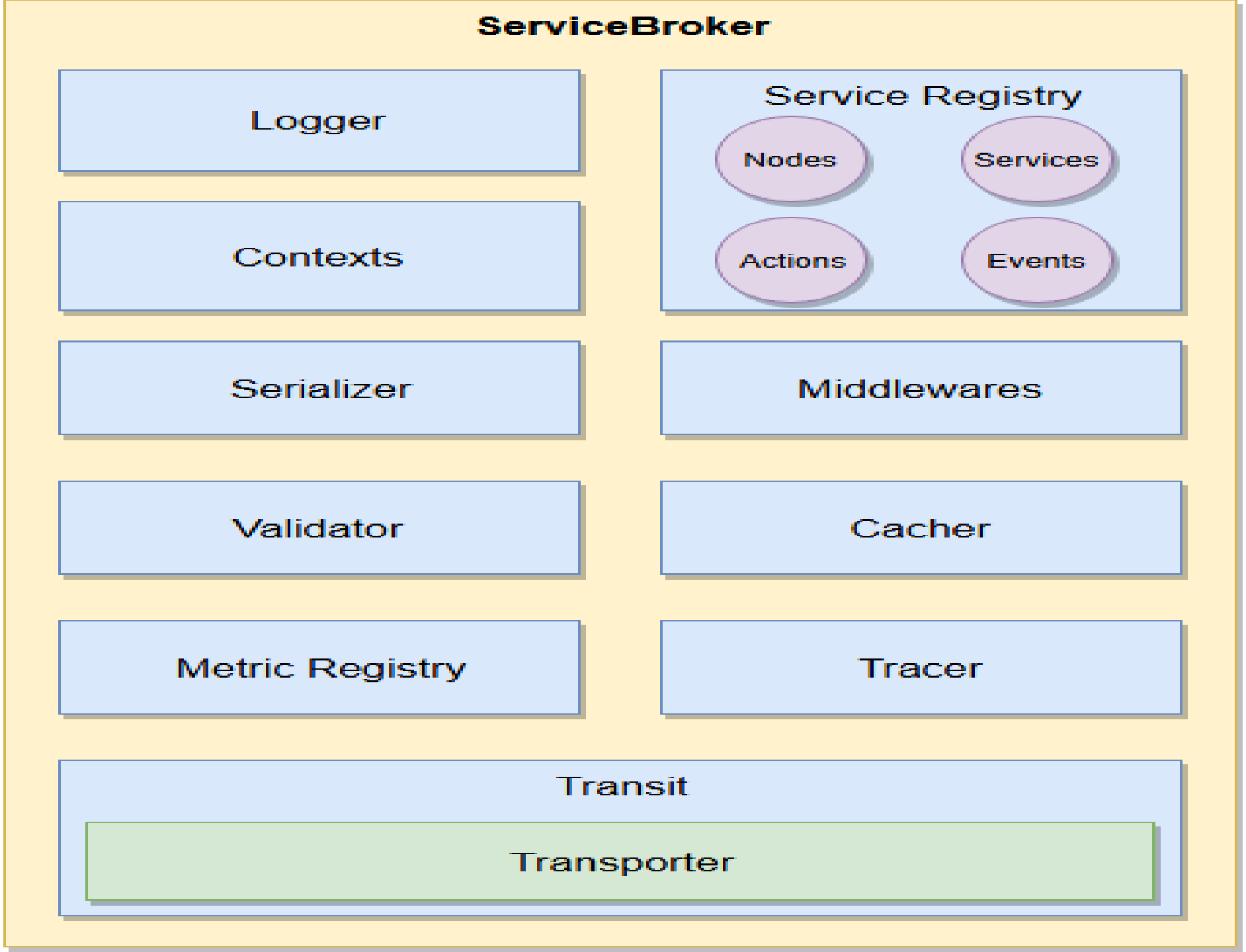
- Online store can be seen as a composition of 2 independent services: the products service and the gateway service.
- First one is responsible for storage and management of the products while the second simply receives user´s requests and conveys them to the products service.
- To ensure that our system is resilient to failures run the products and the gateway services in dedicated nodes (node-1 and node-2).
- Running services at dedicated nodes means that the transporter module is required for inter services communication.
- Most of the transporters rely on a message broker for inter services communication, so we're going to need one up and running.

Architecture

- Request (GET /products) is received by the HTTP server running at node-1.
- Incoming request is simply passed from the HTTP server to the gateway service that does all the processing and mapping.
- User´s request is mapped into a listProducts action of the products service.
- Request is passed to the broker, which checks whether the products service is a local or a remote service.
- Products service is remote so the broker needs to use the transporter module to deliver the request.
- Transporter simply grabs the request and sends it through the communication bus.
- Since both nodes (node-1 and node-2) are connected to the same communication bus (message broker), the request is successfully delivered to the node-2.
- Upon reception, the broker of node-2 will parse the incoming request and forward it to the products service.
- Finally, the products service invokes the listProducts action and returns the list of all available products. The response is simply forwarded back to the end-user.

Broker

- ServiceBroker is the main component of Moleculer.
- Handles services, calls actions, emits events and communicates with remote nodes.
- Must create a ServiceBroker instance on every node.


ServiceBroker
Logger
Contexts
Service Registry
Nodes
Services
Actions
Events
Serializer
Middlewares
Validator
Cacher
Metric Registry
Tracer
Transit
Transporter

Create broker with default settings:

```
const { ServiceBroker } = require("moleculer");
const broker = new ServiceBroker();
```

Create broker with custom settings:

```
const { ServiceBroker } = require("moleculer");
const broker = new ServiceBroker({
    nodeID: "my-node"
});
```

Create broker with transporter to communicate with remote nodes:

```
const { ServiceBroker } = require("moleculer");
const broker = new ServiceBroker({
    nodeID: "node-1",
    transporter: "nats://localhost:4222",
    logLevel: "debug",
    requestTimeout: 5 * 1000
});
```

Metadata option

Use `metadata` property to store custom values. It can be useful for a custom middleware or strategy.

```
const broker = new ServiceBroker({
    nodeID: "broker-2",
    transporter: "NATS",
    metadata: {
        region: "eu-west1"
    }
});
```

Global error handler

Catch, handle & log the error

```
const broker = new ServiceBroker({
    errorHandler(err, info) {
        // Handle the error
        this.logger.warn("Error handled:", err);
    }
});
```

Catch & throw further the error

```
const broker = new ServiceBroker({
    errorHandler(err, info) {
        this.logger.warn("Log the error:", err);
        throw err; // Throw further
    }
});
```

Properties of ServiceBroker

Name	Type	Description
broker.options	Object	Broker options.
broker.Promise	Promise	Bluebird Promise class.
broker.started	Boolean	Broker state.
broker.namespace	String	Namespace.
broker.nodeID	String	Node ID.
broker.instanceID	String	Instance ID.
broker.metadata	Object	Metadata from broker options.
broker.logger	Logger	Logger class of ServiceBroker.
broker.cacher	Cacher	Cacher instance
broker.serializer	Serializer	Serializer instance.

Properties of ServiceBroker

Name	Type	Description
broker.validator	Any	Parameter Validator instance.
broker.services	Array<Service>	Local services.
broker.metrics	MetricRegistry	Built-in Metric Registry.
broker.tracer	Tracer	Built-in Tracer instance.
broker.errorRegenerator	Regenerator	Built-in Regenerator instance.

Methods of ServiceBroker

Name	Response	Description
broker.start()	Promise	Start broker.
broker.stop()	Promise	Stop broker.
broker.repl()	-	Start REPL mode.
broker.errorHandler(err, info)	-	Call the global error handler.
broker.getLogger(module, props)	Logger	Get a child logger.
broker.fatal(message, err, needExit)	-	Throw an error and exit the process.
broker.loadServices(folder, fileMask)	Number	Load services from a folder.
broker.loadService(filePath)	Service	Load a service from file.

Methods of ServiceBroker

Name	Response	Description
broker.createService(schema, schemaMods)	Service	Create a service from schema.
broker.destroyService(service)	Promise	Destroy a loaded local service.
broker.getLocalService(name)	Service	Get a local service instance by full name (e.g. v2.posts)
broker.waitForServices(serviceNames, timeout, interval)	Promise	Wait for services.
broker.call(actionName, params, opts)	Promise	Call a service.
broker.mcall(def)	Promise	Multiple service calling.
broker.emit(eventName, payload, opts)	-	Emit a balanced event.
broker.broadcast(eventName, payload, opts)	-	Broadcast an event.

Methods of ServiceBroker

Name	Response	Description
broker.broadcastLocal(eventName, payload, opts)	-	Broadcast an event to local services only.
broker.ping(nodeID, timeout)	Promise	Ping remote nodes.
broker.hasEventListener("eventName")	Boolean	Checks if broker is listening to an event.
broker.getEventListeners("eventName")	Array<Object>	Returns all registered event listeners for an event name.
broker.generateUid()	String	Generate an UUID/token.
broker.callMiddlewareHook(name, args, opts)	-	Call an async hook in the registered middlewares.
broker.callMiddlewareHookSync(name, args, opts)	-	Call a sync hook in the registered middlewares.
broker.isMetricsEnabled()	Boolean	Check the metrics feature is enabled.
broker.isTracingEnabled()	Boolean	Check the tracing feature is enabled.

Broker options

- Options can be used in ServiceBroker constructor or in moleculer.config.js file.
- Example Object

Broker options

- namespace: String - Namespace of nodes to segment your nodes on the same network (e.g.: “development”, “staging”, “production”). Default: ""
- nodeID: String - Unique node identifier. Must be unique in a namespace. If not the broker will throw a fatal error and stop the process. Default: hostname + PID
- logger: Boolean | String | Object | Array<Object>) - Logger class. By default, it prints message to the console. Read more. _Default: "Console"
- logLevel: String | Object - Log level for loggers (trace, debug, info, warn, error, fatal). Read more. Default: info
- transporter: String | Object | Transporter - Transporter configuration. Read more. Default: null
- requestTimeout: Number - Number of milliseconds to wait before reject a request with a RequestTimeout error. Disabled: 0 Default: 0
- retryPolicy: Object - Retry policy configuration. Read more.
- contextParamsCloning: Boolean - Cloning the params of context if enabled. High performance impact. Use it with caution! Default: false

Broker options

- dependencyInterval: Configurable interval (defined in ms) that's used by the services while waiting for dependency services. Default: 1000
- maxCallLevel: Number - Limit of calling level. If it reaches the limit, broker will throw an MaxCallLevelError error. (Infinite loop protection) Default: 0
- heartbeatInterval: Number - Number of seconds to send heartbeat packet to other nodes. Default: 5
- heartbeatTimeout: Number - Number of seconds to wait before setting remote nodes to unavailable status in Registry. Default: 15
- tracking: Object - Tracking requests and waiting for running requests before shutdowning. (Graceful shutdown) Read more.
- disableBalancer: Boolean - Disable built-in request & emit balancer. Transporter must support it, as well. Read more. Default: false
- registry: Object - Settings of Service Registry.
- circuitBreaker: Object - Settings of Circuit Breaker.
- bulkhead: Object - Settings of bulkhead.
- transit.maxQueueSize: Number - A protection against inordinate memory usages when there are too many outgoing requests. If there are more than stated outgoing live requests, the new requests will be rejected with QueueIsFullError error. Default: 50000

Broker options

- transit.maxChunkSize Number - Maximum chunk size while streaming. Default: 256KB
- transit.disableReconnect: Boolean - Disables the reconnection logic while starting a broker. Default: false
- transit.disableVersionCheck: Boolean - Disable protocol version checking logic in Transit. Default: false
- transit.packetLogFilter: Array - Filters out the packets in debug log messages. It can be useful to filter out the HEARTBEAT packets while debugging. Default: []
- uidGenerator: Function - Custom UID generator function for Context ID.
- errorHandler: Function - Global error handler function.
- cacher: String | Object | Cacher - Cacher settings. Read more. Default: null
- serializer: String | Serializer - Instance of serializer. Read more. Default: JSONSerializer
- validator: Boolean | Validator - Enable the default or create custom parameters validation. Default: true

Broker options

- errorRegenerator: Regenerator - Instance of error regenerator. Read more. Default: null
- metrics: Boolean | Object - Enable & configure metrics feature. Default: false
- tracing: Boolean | Object - Enable & configure tracing feature. Default: false
- internalServices: Boolean | Object - Register internal services at start. Default: true
- internalServices.$node - Object - Extend internal services with custom actions. Default: null
- internalMiddlewares: Boolean - Register internal middlewares. Default: true
- hotReload: Boolean - Watch the loaded services and hot reload if they changed. Read more. Default: false
- middlewares: Array<Object> - Register custom middlewares. Default: null
- replDelimiter: String - Custom REPL commands delimiter. Default: mol $
- replCommands: Array<Object> - Register custom REPL commands. Default: null
- metadata: Object - Store custom values. Default: null

Broker options

- skipProcessEventRegistration: Boolean - Skip the default graceful shutdown event handlers. In this case, you have to register them manually. Default: false
- maxSafeObjectSize: Number - Maximum size of objects that can be serialized. On serialization process, check each object property size (based on length or size property value) and trim it, if object size bigger than maxSafeObjectSize value. Default: null
- created: Function - Fired when the broker created. Default: null
- started: Function - Fired when the broker started (all local services loaded & transporter is connected). Default: null
- stopped: Function - Fired when the broker stopped (all local services stopped & transporter is disconnected). Default: null
- ServiceFactory: ServiceClass - Custom Service class. If not null, broker will use it when creating services by service schema. Default: null
- ContextFactory: ContextClass - Custom Context class. If not null, broker will use it when creating contexts for requests & events. Default: null

Services

- Service represents a microservice in the Moleculer framework.
- Can define actions and subscribe to events.
- To create a service you must define a schema.
- Service schema is similar to a component of VueJS.
- Schema has some main parts: name, version, settings, actions, methods, events.

Simple service schema to define two actions

```
// math.service.js
module.exports = {
    name: "math",
    actions: {
        add(ctx) {
            return Number(ctx.params.a) + Number(ctx.params.b);
        },

        sub(ctx) {
            return Number(ctx.params.a) - Number(ctx.params.b);
        }
    }
}
```

Base properties : name

- Name is a mandatory property so it must be defined.
- It's the first part of action name when you call it.
- To disable service name prefixing set $noServiceNamePrefix: true in Service settings.

```
// posts.v1.service.js
module.exports = {
    name: "posts",
    version: 1
}
```

Version

- Version is an optional property.
- Use it to run multiple version from the same service.
- Is a prefix in the action name.
- Can be a Number or a String

```
// posts.v2.service.js
module.exports = {
    name: "posts",
    version: 2,
    actions: {
        find() {...}
    }
}
```

Version

- To call this find action on version 2 service:

broker.call("v2.posts.find");

- REST call --Via API Gateway, make a request to

GET /v2/posts/find.

- To disable version prefixing set $noVersionPrefix: true in Service settings.

Settings

- Settings property is a static store, where you can store every settings/options to your service.
- Can reach it via this.settings inside the service.
- settings is also obtainable on remote nodes.
- Also transferred during service discovering.

Settings

```
// mailer.service.js
module.exports = {
    name: "mailer",
    settings: {
        transport: "mailgun"
    },

    action: {
        send(ctx) {
            if (this.settings.transport == "mailgun") {
                ...
            }
        }
    }
}
```

Internal Settings

Name	Type	Default	Description
$noVersionPrefix	Boolean	false	Disable version prefixing in action names.
$noServiceNamePrefix	Boolean	false	Disable service name prefixing in action names.
$dependencyTimeout	Number	0	Timeout for dependency waiting.
$shutdownTimeout	Number	0	Timeout for waiting for active requests at shutdown.
$secureSettings	Array	[]	List of secure settings.

Mixins

- Mixins are a flexible way to distribute reusable functionalities for Moleculer services.
- Service constructor merges these mixins with the current schema.
- When a service uses mixins, all properties present in the mixin will be “mixed” into the current service.

Example how to extend moleculer-web service

- Example creates an api service which inherits all properties from ApiGwService but overwrite the port setting and extend it with a new myAction action.

```
// api.service.js
const ApiGwService = require("moleculer-web");

module.exports = {
    name: "api",
    mixins: [ApiGwService]
    settings: {
        // Change port setting
        port: 8080
    },
    actions: {
        myAction() {
            // Add a new action to apiGwService service
        }
    }
}
```

Actions

- The actions are the callable/public methods of the service.
- Are callable with broker.call or ctx.call.
- Action could be a Function (shorthand for handler) or an object with some properties and handler.
- Actions should be placed under the actions key in the schema

```
// math.service.js
module.exports = {
    name: "math",
    actions: {
        // Shorthand definition, only a handler function
        add(ctx) {
            return Number(ctx.params.a) + Number(ctx.params.b);
        },

        // Normal definition with other properties. In this case
        // the `handler` function is required!
        mult: {
            cache: false,
            params: {
                a: "number",
                b: "number"
            },
            handler(ctx) {
                // The action properties are accessible as `ctx.action.*`
                if (!ctx.action.cache)
                    return Number(ctx.params.a) * Number(ctx.params.b);
            }        }        }  };
```

Actions

- Can call the actions as

```
const res = await broker.call("math.add", { a: 5, b: 7 });
```

```
const res = await broker.call("math.mult", { a: 10, b: 31 });
```

- Inside actions, you can call other nested actions in other services with ctx.call method.
- ctx.call is an alias to broker.call, but it sets itself as parent context (due to correct tracing chains).

```
// posts.service.js
module.exports = {
    name: "posts",
    actions: {
        async get(ctx) {
            // Find a post by ID
            let post = posts[ctx.params.id];

            // Populate the post.author field through "users" service
            // Call the "users.get" action with author ID
            const user = await ctx.call("users.get", { id: post.author });
            if (user) {
                // Replace the author ID with the received user object
                post.author = user;
            }

            return post;
        }
    }
};
```

Events

- can subscribe to events under the events key

```
// report.service.js
module.exports = {
name: "report",

events: {
// Subscribe to "user.created" event
"user.created"(ctx) {
this.logger.info("User created:", ctx.params);
// Do something
},

// Subscribe to all "user.*" events
"user.*"(ctx) {
console.log("Payload:", ctx.params);
console.log("Sender:", ctx.nodeID);
console.log("Metadata:", ctx.meta);
console.log("The called event name:", ctx.eventName);
}
// Subscribe to a local event
"$node.connected"(ctx) {
this.logger.info(`Node '${ctx.params.id}' is connected!`);
} } };
```

Methods

- To create private methods in the service, put your functions under the methods key.
- These functions are private, can't be called with broker.call.
- But you can call it inside service (from action handlers, event handlers and lifecycle event handlers).

```
// mailer.service.js
module.exports = {
    name: "mailer",
    actions: {
        send(ctx) {
            // Call the `sendMail` method
            return this.sendMail(ctx.params.recipients, ctx.params.subject,
ctx.params.body);
        }
    },

    methods: {
        // Send an email to recipients
        sendMail(recipients, subject, body) {
            return new Promise((resolve, reject) => {
                ...
            });
        }
    }
};
```

Lifecycle Events

- There are some lifecycle service events, that will be triggered by broker.
- Are placed in the root of schema

```
// www.service.js
module.exports = {
name: "www",
actions: {...},
events: {...},
methods: {...},

created() {
// Fired when the service instance created (with `broker.loadService` or `broker.createService`)
},

merged() {
// Fired after the service schemas merged and before the service instance created
},

async started() {
// Fired when broker starts this service (in `broker.start()`)
}

async stopped() {
// Fired when broker stops this service (in `broker.stop()`)
}
};
```

Dependencies

- If your service depends on other services, use the dependencies property in the schema.
- The service waits for dependent services before calls the started lifecycle event handler.
- The started service handler is called once the likes, v2.auth, v2.users, staging.comments services are available (either the local or remote nodes).

```
// posts.service.js
module.exports = {
name: "posts",
settings: {
$dependencyTimeout: 30000 // Default: 0 - no timeout
},
dependencies: [
"likes", // shorthand w/o version
"v2.auth", // shorthand w version
{ name: "users", version: 2 }, // with numeric version
{ name: "comments", version: "staging" } // with string version
],
async started() {
this.logger.info("It will be called after all dependent services are available.");
const users = await this.broker.call("users.list");
}
....
}
```

Wait for services via ServiceBroker

- To wait for services, can also use the waitForServices method of ServiceBroker.
- Returns a Promise which will be resolved, when all defined services are available & started.

Parameter	Type	Default	Description
services	String or Array	-	Service list to waiting
timeout	Number	0	Waiting timeout. 0 means no timeout. If reached, a MoleculerServerError will be rejected.
interval	Number	1000	Frequency of watches in milliseconds

Wait for services via ServiceBroker

Example

```
broker.waitForServices(["posts", "v2.users"]).then(() => {
    // Called after the `posts` & `v2.users` services are available
});
```

Set timeout & interval

```
broker.waitForServices("accounts", 10 * 1000, 500).then(() => {
    // Called if `accounts` service becomes available in 10 seconds
}).catch(err => {
    // Called if service is not available in 10 seconds
});
```

Metadata

- Service schema has a metadata property.
- Can store here any meta information about service.
- Can access it as **this.metadata** inside service functions.
- Moleculer core modules don't use it. Can store in it whatever you want.

```
module.exports = {
    name: "posts",
    settings: {},
    metadata: {
        scalable: true,
        priority: 5
    },

    actions: { ... }
};
```

Properties of Service Instances

Name	Type	Description
this.name	String	Name of service (from schema)
this.version	Number or String	Version of service (from schema)
this.fullName	String	Name of version prefix
this.settings	Object	Settings of service (from schema)
this.metadata	Object	Metadata of service (from schema)
this.schema	Object	Schema definition of service
this.broker	ServiceBroker	Instance of broker

Properties of Service Instances

Name	Type	Description
this.Promise	Promise	Class of Promise (Bluebird)
this.logger	Logger	Logger instance
this.actions	Object	Actions of service. *Service can call own actions directly*
this.waitForServices	Function	Link to broker.waitForServices method
this.currentContext	Context	Get or set the current Context object.

Service Creation-- broker.createService()

```
broker.createService({
    name: "math",
    actions: {
        add(ctx) {
            return Number(ctx.params.a) + Number(ctx.params.b);
        }
    }
});
```

Service Creation-- Load service from file

math.service.js

```
// Export the schema of service
module.exports = {
    name: "math",
    actions: {
        add(ctx) {
            return Number(ctx.params.a) + Number(ctx.params.b);
        },
        sub(ctx) {
            return Number(ctx.params.a) - Number(ctx.params.b);
        }
    }
}
```

Load it with broker:

```
// Create broker
const broker = new ServiceBroker();

// Load service
broker.loadService("./math.service");

// Start broker
broker.start();
```

Load multiple services from a folder

Syntax

```
broker.loadServices(folder = "./services", fileMask = "**/*.service.js");
```

Example

```
// Load every *.service.js file from the "./services" folder (including subfolders)
broker.loadServices();

// Load every *.service.js file from the current folder (including subfolders)
broker.loadServices("./");

// Load every user*.service.js file from the "./svc" folder
broker.loadServices("./svc", "user*.service.js");
```

Load with Moleculer Runner

- Moleculer Runner is a helper script that helps you run Moleculer projects.
- With it, don't need to create a ServiceBroker instance with options.
- Instead, can create a moleculer.config.js file in the root of repo with broker options.
- Then simply call the moleculer-runner in NPM script, and it will automatically load the configuration file, create the broker and load the services.
- Alternatively, can declare your configuration as environment variables.

moleculer-runner [options] [service files or directories or glob masks]

Moleculer runner Options

Option	Type	Default	Description
-r, --repl	Boolean	false	If true, it switches to REPL mode after broker started.
-s, --silent	Boolean	false	Disable the broker logger. It prints nothing to the console.
-H, --hot	Boolean	false	Hot reload services when they change.
-c, --config <file>	String	null	Load configuration file from a different path or a different filename.
-e, --env	Boolean	false	Load environment variables from the '.env' file from the current folder.
-E, --envfile <file>	String	null	Load environment variables from the specified file.
-i, --instances	Number	null	Launch [number] node instances or max for all cpu cores (with cluster module)

Moleculer runner

Example NPM scripts

```
{
    "scripts": {
        "dev": "moleculer-runner --repl --hot --config moleculer.dev.config.js services",
        "start": "moleculer-runner --instances=max services"
    }
}
```

The `dev` script loads development configurations from the `moleculer.dev.config.js` file, start all services from the `services` folder, enable hot-reloading and switches to REPL mode. Run it with the `npm run dev` command.

The `start` script is to load the default `moleculer.config.js` file if it exists, otherwise only loads options from environment variables. Starts 4 instances of broker, then they start all services from the `services` folder. Run it with `npm start` command.

Configuration loading logic

Runner does the following steps to load & merge configurations:

1. Load the config file defined in MOLECULER_CONFIG environment variable.
   - If it does not exist, it throws an error.
2. It loads config file defined in CLI options.
   - If it does not exist, it throws an error.
   - Note that MOLECULER_CONFIG has priority over CLI meaning that if both are defined MOLECULER_CONFIG is the one that's going to be used.
3. If not defined, it loads the moleculer.config.js file from the current directory.
   - If it does not exist, it loads the moleculer.config.json file.
4. Once a config file has been loaded, it merges options with the default options of the ServiceBroker.
5. The runner observes the options step by step and tries to overwrite them from environment variables.

Configuration file

- Structure of the configuration file is the same as that of the broker options.
- Every property has the same name.
- Asynchronous Configuration file --Moleculer Runner also supports asynchronous configuration files. In this case moleculer.config.js must export a Function that returns a Promise (or you can use async/await).

Example config file

```
// moleculer.config.js
module.exports = {
nodeID: "node-test",
logger: true,
logLevel: "debug",

transporter: "nats://localhost:4222",
requestTimeout: 5 * 1000,

circuitBreaker: {
enabled: true
},

metrics: true
};
```

Example config file -- async

```
// moleculer.config.js
const fetch = require("node-fetch");

module.exports = async function() {
const res = await fetch("https://pastebin.com/raw/SLZRqfHX");
return await res.json();
};
```

Built-in clustering

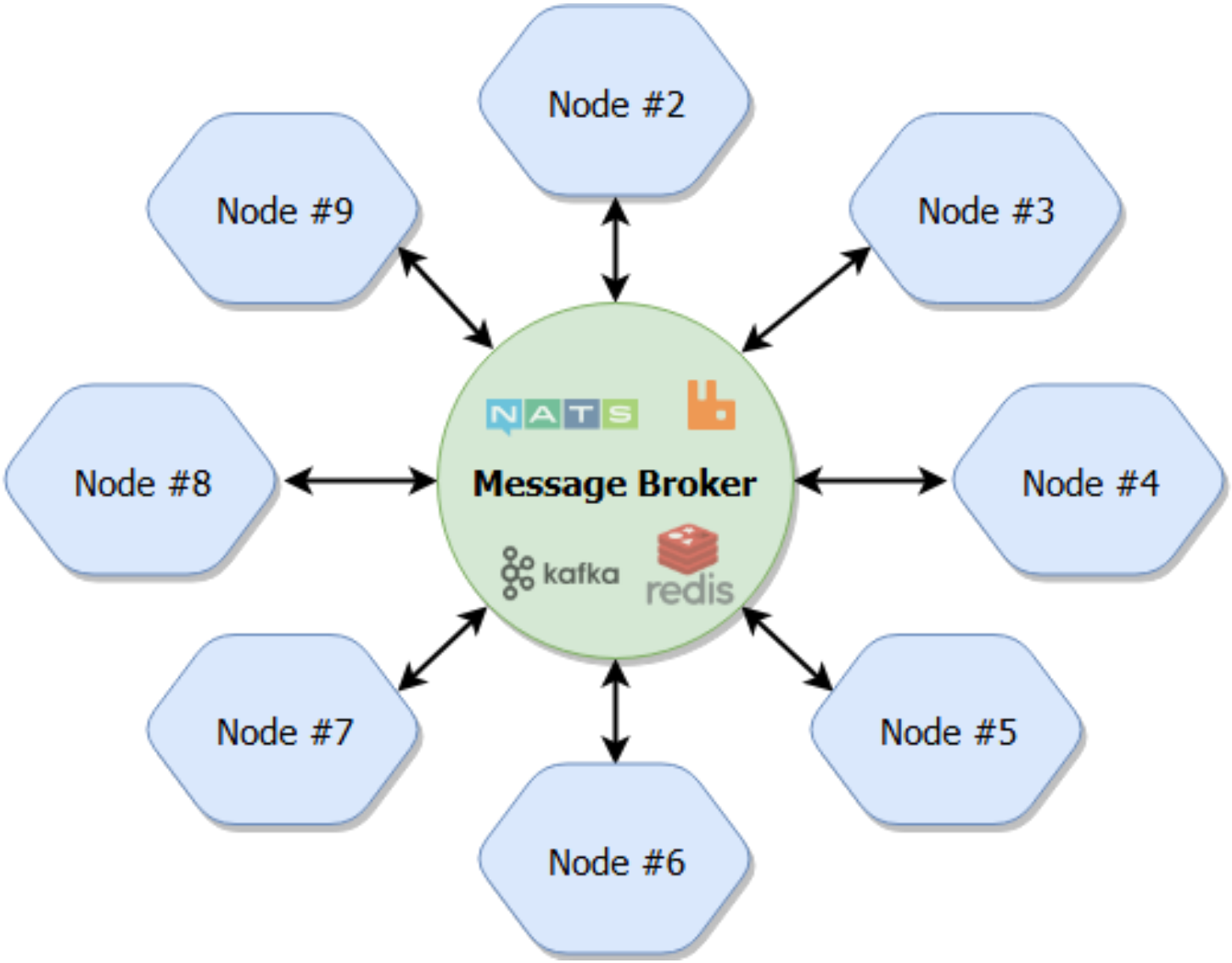
- Moleculer Runner has a built-in clustering function to start multiple instances from your broker.
- Example to start all services from the services folder in 4 instances.

$ moleculer-runner --instances 4 services

- The nodeID will be suffixed with the worker ID.
- E.g. if you define my-node nodeID in options, and starts 4 instances, the instance nodeIDs will be my-node-1, my-node-2, my-node-3, my-node-4.

Networking

- To communicate with other nodes (ServiceBrokers) need to configure a transporter.
- Most of the supported transporters connect to a central message broker that provide a reliable way of exchanging messages among remote nodes.
- Message brokers mainly support publish/subscribe messaging pattern.


Node #2
Node #9
Node #3
NATS
Node #8
Message Broker
Node #4
kafka
redis
Node #7
Node #5
Node #6

Transporters

- Transporter is an important module if you are running services on multiple nodes.
- Transporter communicates with other nodes.
- Transfers events, calls requests and processes responses …etc. If multiple instances of a service are running on different nodes then the requests will be load-balanced among them.
- Whole communication logic is outside of transporter class.
- **Means that you can switch between transporters without changing any line of code.**

Transporters

- Several built-in transporters in Moleculer framework

1. TCP transporter
2. NATS Transporter
3. Redis Transporter
4. MQTT Transporter
5. AMQP (0.9) Transporter
6. AMQP 1.0 Transporter
7. Kafka Transporter
8. NATS Streaming (STAN) Transporter
9. Cus tom transporter

Serialization

- Transporter needs a serializer module which serializes & deserializes the transferred packets.
- The default serializer is the **JSONSerializer**
  - Serializes the packets to JSON string and deserializes the received data to packet.
- Other built-in serializer :

1. Avro serializer
2. MsgPack serializer
3. Notepack serializer
4. ProtoBuf serializer
5. Thrift serializer
6. CBOR serializer
7. Custom serializer

Fault tolerance

- Moleculer has several built-in fault-tolerance features.
- Can be enabled or disabled in broker options.
- **Circuit Breaker**
  - Moleculer has a built-in circuit-breaker solution.
  - Threshold-based implementation.
  - Uses a time window to check the failed request rate. Once the threshold value is reached, it trips the circuit breaker.

circuit breaker

- Can prevent an application from repeatedly trying to execute an operation that's likely to fail.
- Allowing it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long lasting.
- Circuit Breaker pattern also enables an application to detect whether the fault has been resolved.
- If the problem appears to have been fixed, the application can try to invoke the operation.
- **If you enable it, all service calls will be protected by the circuit breaker.**

circuit breaker

Enable it in the broker options

```
const broker = new ServiceBroker({
    circuitBreaker: {
        enabled: true,
        threshold: 0.5,
        minRequestCount: 20,
        windowTime: 60, // in seconds
        halfOpenTime: 5 * 1000, // in milliseconds
        check: err => err && err.code >= 500
    }
});
```

Global options can be overridden in action definition

```
// users.service.js
module.export = {
    name: "users",
    actions: {
        create: {
            circuitBreaker: {
                // All CB options can be overwritten from broker options.
                threshold: 0.3,
                windowTime: 30
            },
            handler(ctx) {}
        }
    }
};
```

circuit breaker -settings

Name	Type	Default	Description
enabled	Boolean	false	Enable feature
threshold	Number	0.5	Threshold value. 0.5 means that 50% should be failed for tripping.
minRequestCount	Number	20	Minimum request count. Below it, CB does not trip.
windowTime	Number	60	Number of seconds for time window.
halfOpenTime	Number	10000	Number of milliseconds to switch from open to half-open state
check	Function	err && err.code >= 500	A function to check failed requests.

Retry

- There is an exponential backoff retry solution.
- **Can recall failed requests with response MoleculerRetryableError.**

Enable retry in broker options

```
const broker = new ServiceBroker({
    retryPolicy: {
        enabled: true,
        retries: 5,
        delay: 100,
        maxDelay: 2000,
        factor: 2,
        check: err => err && !!err.retryable
    }
});
```

Overwrite the retries value in calling option

```
broker.call("posts.find", {}, { retries: 3 });
```

Overwrite the retry policy values in action definitions

```
// users.service.js
module.export = {
    name: "users",
    actions: {
        find: {
            retryPolicy: {
                // All Retry policy options can be overwritten from broker options.
                retries: 3,
                delay: 500
            },
            handler(ctx) {}
        },
        create: {
            retryPolicy: {
                // Disable retries for this action
                enabled: false
            },
            handler(ctx) {}
        }
    }
};
```

Retry option settings

Name	Type	Default	Description
enabled	Boolean	false	Enable feature.
retries	Number	5	Count of retries.
delay	Number	100	First delay in milliseconds.
maxDelay	Number	2000	Maximum delay in milliseconds.
factor	Number	2	Backoff factor for delay. 2 means exponential backoff.
check	Function	err && !!err.retryable	A function to check failed requests.

Timeout

- Timeout can be set for service calling.
- Can be set globally in broker options, or in calling options.
- If the timeout is defined and request is timed out, broker will throw a RequestTimeoutError error.

Enable it in the broker options

```
const broker = new ServiceBroker({
    requestTimeout: 5 * 1000 // in milliseconds
});
```

Overwrite the timeout value in calling option

```
broker.call("posts.find", {}, { timeout: 3000 });
```

Fallback

- Fallback feature is useful, when you don't want to give back errors to the users.
- Instead, call an other action or return some common content.
- Fallback response can be set in calling options or in action definition.
- Should be a Function which returns a Promise with any content.
- Broker passes the current Context & Error objects to this function as arguments.

- Fallback response can be also defined in receiver-side, in action definition.

Fallback response setting in calling options

```
const result = await broker.call("users.recommendation", { userID: 5 }, {
    timeout: 500,
    fallbackResponse(ctx, err) {
        // Return a common response from cache
        return broker.cacher.get("users.fallbackRecommendation:" + ctx.params.userID);
    }
});
```

Fallback as a function

```
module.exports = {
    name: "recommends",
    actions: {
        add: {
            fallback: (ctx, err) => "Some cached result",
            handler(ctx) {
                // Do something
            }
        }
    }
};
```

Fallback as method name string

```
module.exports = {
    name: "recommends",
    actions: {
        add: {
            // Call the 'getCachedResult' method when error occurred
            fallback: "getCachedResult",
            handler(ctx) {
                // Do something
            }
        }
    },

    methods: {
        getCachedResult(ctx, err) {
            return "Some cached result";
        }
    }
};
```