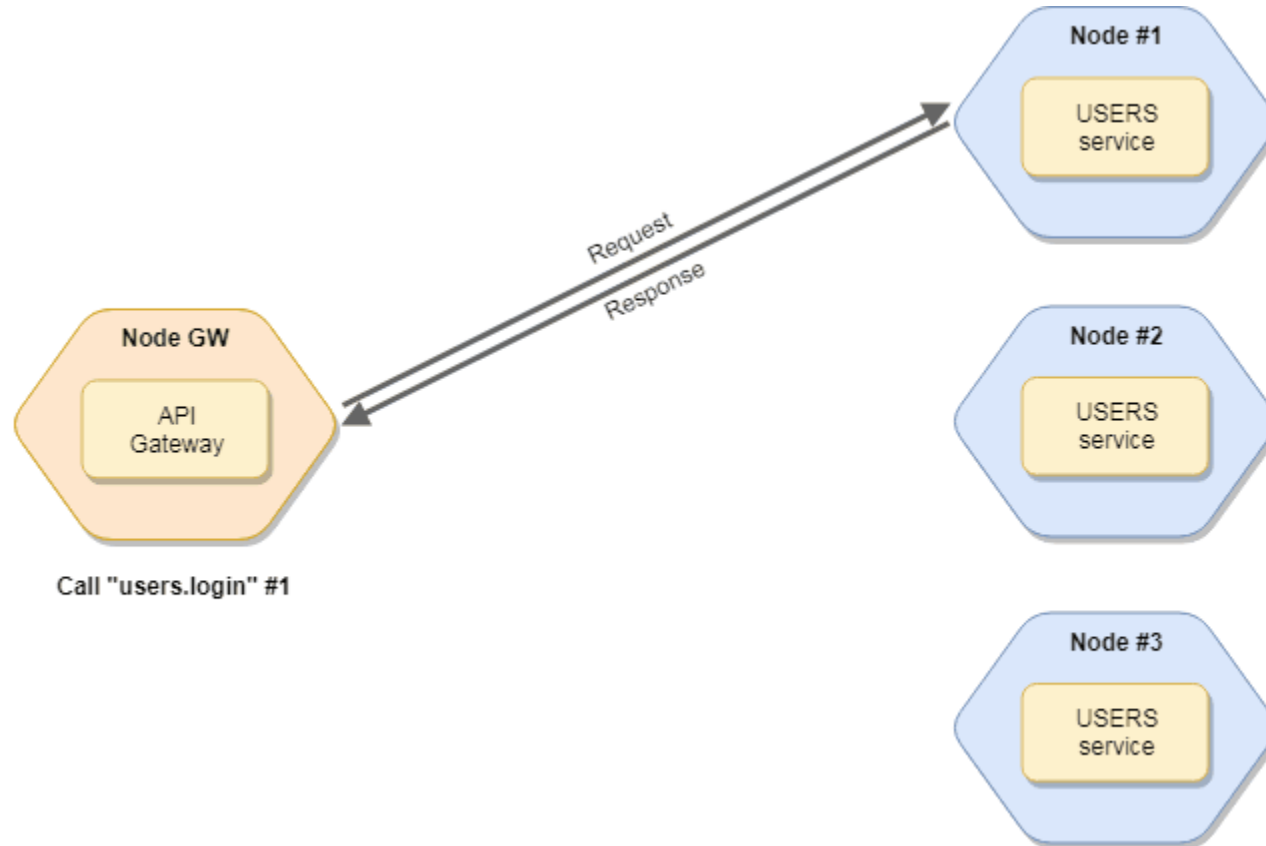


Actions in molecular framework

ANJU MUNOTH



Actions

- ▶ Actions are the callable/public methods of the service.
- ▶ Action calling represents a remote-procedure-call (RPC).
- ▶ Has request parameters & returns response, like a HTTP request.
- ▶ If you have multiple instances of services, the broker will load balance the request among instances.

Call services

- ▶ To call a service use the `broker.call` method.
- ▶ Broker looks for the service (and a node) which has the given action and call it.
- ▶ Function returns a Promise.

Syntax

`const res = await broker.call(actionName, params, opts);`

- ▶ ActionName is a dot-separated string.
- ▶ The first part of it is the service name, while the second part of it represents the action name.
- ▶ A posts service with a create action --call it as **`posts.create`**.
- ▶ Params is an object which is passed to the action as a part of the Context.
- ▶ Service can access it via `ctx.params`. It is optional. If you don't define, it will be `{}`.
- ▶ Opts is an object to set/override some request parameters, e.g.: `timeout`, `retryCount`. It is optional.

Available calling options:

Name	Type	Default	Description
timeout	Number	null	Timeout of request in milliseconds. If the request is timed out and you don't define fallbackResponse, broker will throw a RequestTimeout error. To disable set 0. If it's not defined, the requestTimeout value from broker options will be used.
retries	Number	null	Count of retry of request. If the request is timed out, broker will try to call again. To disable set 0. If it's not defined, the retryPolicy.retries value from broker options will be used.
fallbackResponse	Any	null	Returns it, if the request has failed.

Available calling options:

Name	Type	Default	Description
nodeID	String	null	Target nodeID. If set, it will make a direct call to the specified node.
meta	Object	{}	Metadata of request. Access it via ctx.meta in actions handlers. It will be transferred & merged at nested calls, as well.
parentCtx	Context	null	Parent Context instance. Use it to chain the calls.
requestID	String	null	Request ID or Correlation ID. Use it for tracing.

Actions -- call

Call without params

```
const res = await broker.call("user.list");
```

Call with params

```
const res = await broker.call("user.get", { id: 3 });
```

Call with calling options

```
const res = await broker.call("user.recommendation", { limit: 5 }, {  
  timeout: 500,  
  retries: 3,  
  fallbackResponse: defaultRecommendation  
});
```

Actions -- call

Call with promise error handling

```
broker.call("posts.update", { id: 2, title: "Modified post title" })  
  .then(res => console.log("Post updated!"))  
  .catch(err => console.error("Unable to update Post!", err));
```

Direct call: get health info from the “node-21” node

```
const res = await broker.call("$node.health", null, { nodeID: "node-21" })
```


Metadata

```
broker.createService({
  name: "test",
  actions: {
    first(ctx) {
      return ctx.call("test.second", null, { meta: {
        b: 5
      }});
    },
    second(ctx) {
      console.log(ctx.meta);
      // Prints: { a: "John", b: 5 }
    }
  }
});

broker.call("test.first", null, { meta: {
  a: "John"
}});
```

Multiple calls

- ▶ Calling multiple actions at the same time is also possible. To do it use `broker.mcall` or `ctx.mcall`.

```
await broker.mcall(  
  [  
    { action: 'posts.find', params: { author: 1 }, options: { /* Calling options for this call. */ },  
    { action: 'users.find', params: { name: 'John' } }  
  ],  
  {  
    // Common calling options for all calls.  
    meta: { token: '63f20c2d-8902-4d86-ad87-b58c9e2333c2' }  
  }  
);
```

Action visibility

- ▶ The action has a visibility property to control the visibility & callability of service actions.
- ▶ **published or null:** public action. It can be called locally, remotely and can be published via API Gateway
- ▶ **public:** public action, can be called locally & remotely but not published via API GW
- ▶ **protected:** can be called only locally (from local services)
- ▶ **private:** can be called only internally (via `this.actions.xy()` inside service)

Change visibility

```
module.exports = {
  name: "posts",
  actions: {
    // It's published by default
    find(ctx) {},
    clean: {
      // Callable only via `this.actions.clean`
      visibility: "private",
      handler(ctx) {}
    }
  },
  methods: {
    cleanEntities() {
      // Call the action directly
      return this.actions.clean();
    }
  }
}
```

Action hooks

- ▶ Action hooks are pluggable and reusable middleware functions that can be registered before, after or on errors of service actions.
- ▶ A hook is either a Function or a String.
- ▶ In case of a String it must be equal to service's method name.

Before hooks

- ▶ In before hooks, it receives the ctx, it can manipulate the ctx.params, ctx.meta, or add custom variables into ctx.locals what you can use in the action handlers.
- ▶ If there are any problem, it can throw an Error.
- ▶ Can't break/skip the further executions of hooks or action handler.

Main usages:

- ▶ parameter sanitization
- ▶ parameter validation
- ▶ entity finding
- ▶ authorization

After hooks

- ▶ In after hooks, it receives the ctx and the response.
- ▶ Can manipulate or completely change the response.
- ▶ In the hook, it has to return the response.

Main usages:

- ▶ property populating
- ▶ remove sensitive data.
- ▶ wrapping the response into an Object
- ▶ convert the structure of the response

Error hooks

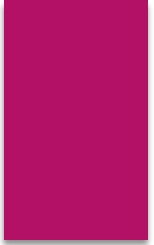
- ▶ The error hooks are called when an Error is thrown during action calling.
- ▶ Receives the ctx and the err.
- ▶ Can handle the error and return another response (fallback) or throws further the error.

Main usages:

- ▶ error handling
- ▶ wrap the error into another one
- ▶ fallback response

Reusability

- ▶ The most efficient way of reusing hooks is by declaring them as service methods in a separate file and import them with the mixin mechanism.
- ▶ This way a single hook can be easily shared across multiple actions.



```
// authorize.mixin.js
module.exports = {
  methods: {
    checkIsAuthenticated(ctx) {
      if (!ctx.meta.user)
        throw new Error("Unauthenticated");
    },
    checkUserRole(ctx) {
      if (ctx.action.role && ctx.meta.user.role !== ctx.action.role)
        throw new Error("Forbidden");
    },
    checkOwner(ctx) {
      // Check the owner of entity
    }
  }
}
```

```
// posts.service.js
const MyAuthMixin = require("../authorize.mixin");
module.exports = {
  name: "posts",
  mixins: [MyAuthMixin]
  hooks: {
    before: {
      "*": ["checkIsAuthenticated"],
      create: ["checkUserRole"],
      update: ["checkUserRole", "checkOwner"],
      remove: ["checkUserRole", "checkOwner"]
    }
  },
  actions: {
    find: { // No required role
      handler(ctx) {}
    },
    create: {
      role: "admin", handler(ctx) {}
    },
    update: {
      role: "user", handler(ctx) {}
    }
  }
};
```





