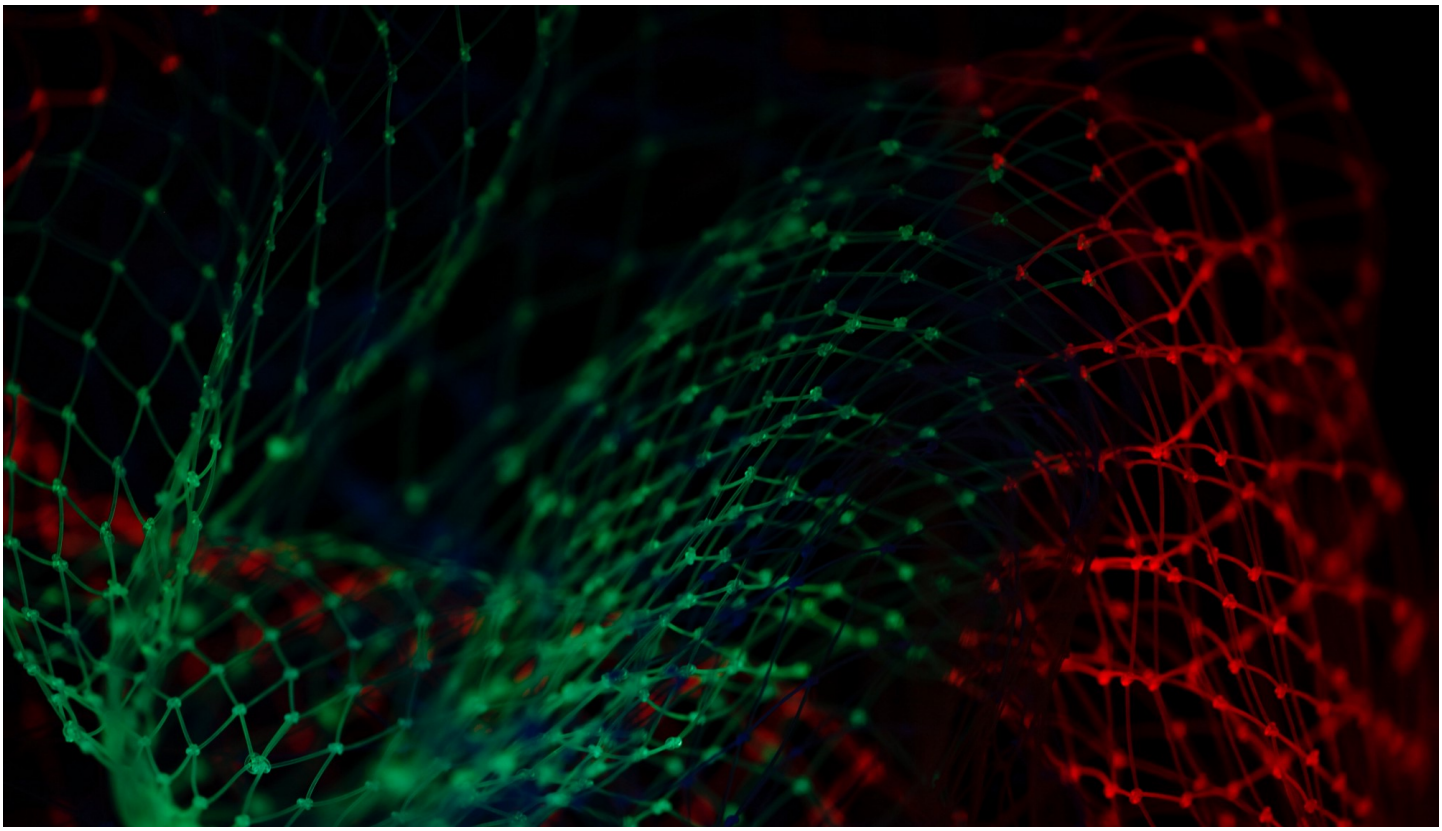
Design patterns for modern web APIs

David Luecke
Nov 18, 2018 · 10 min read

For the past decade, much of my professional life has evolved around making applications and (web)servers talk to each other. After making several contributions to one of the first client side JavaScript frameworks in 2007, I became inspired by the idea of small, data-driven web APIs. Three years later, I had the opportunity to research and implement a project that allowed to make an application available through different remote procedure call (RPC) protocols as my university final thesis.

Since then I had the chance to use and refine many of the patterns I researched during that time in many different environments, from private projects over startups to some of the biggest technology companies in the world and in many different programming languages from Java and Groovy over Python and PHP to NodeJS, where Feathers is the latest distillation of this journey.



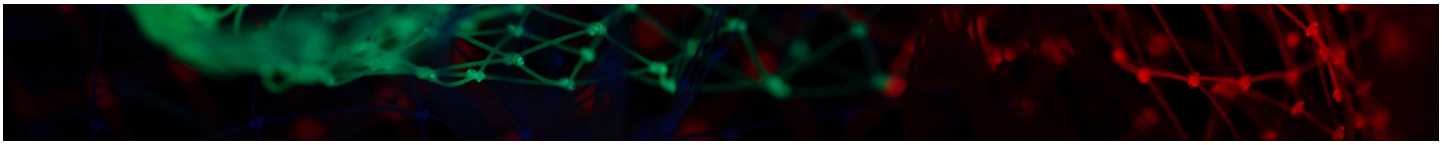


Photo by Pietro Jeng on Unsplash

More important than the choice of programming language or framework however are the design patterns that we can use to help us create software:

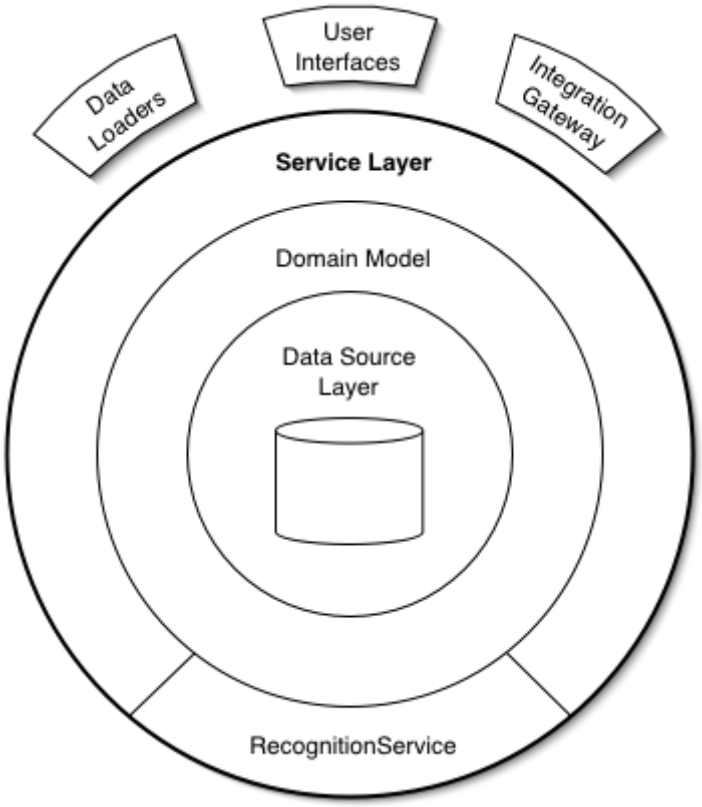
> *Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.*

In this post, I'd like to look at a combination of high level design and architectural patterns that I believe can help create web APIs that are more flexible, maintainable and easier to understand in any programming language.

Service layer

The service layer is a common interface to your application logic that different clients like a web interface, a command line tool or a scheduled job can use. According to Patterns Of Enterprise application architecture the service layer

> Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.



The Service Layer (source: P of EAA catalog)

In its basic form, a service layer provides a set of methods that any other client can use:

```
class ChatApplication {
  login (username, password) {},

  logout (user) {},

  joinChatRoom(roomId, user) {}

  sendMessage(message, roomId, user) {}

  sendPrivateMessage(message, receiverId, user) {}
}
```

app.js hosted with ♡ by GitHub view raw

The service layer methods itself then implement the application logic and make calls to the databases or models. It does not have to be a class but can also be a set of functions that are publicly exposed. In the context of web APIs, both ways offer several advantages:

- **Separation of Concerns and Testability**: Instead of implementing all functionality directly e.g. in the controller actions or route handlers of a web framework, the service layer allows to test the application logic separate from how it is being accessed. A CLI, GUI or web framework that uses the service layer then only needs integration tests instead of having to test the integration and application logic at the same time.
- **Self documenting**: New developers can see what the application provides in the context of the programming language (classes or functions that can be called) instead of having to read often outdated documentation or reverse-engineer external HTTP API interfaces.
- **Transport agnostic**: Different clients, like a message queue integration or a scheduled job can use the application directly or through a more appropriate transport protocol without being forced to use HTTP.

REST

Representational State Transfer (REST) is an architectural style that defines a set of constraints for creating web APIs. The term was coined by Roy Fielding in his PHD thesis and expands many of the design decisions that went into the HTTP protocol into a more high level architecture. Although it is most often used in the context of HTTP, REST is an architectural design pattern and not a communication protocol. As one implementation of the REST architecture, HTTP is considered a RESTful protocol.

Constraints

Since it is not a formally defined protocol there are many opinions on the details of implementing REST APIs. However, the following five constraints must be present for any application to be considered RESTful:

- **Client-server**: A client-server architecture allows a clear separation of concerns. The client is responsible for requesting and displaying the data while the server is taking care of data storage and application logic. One advantage is that both sides can be developed separately as long as the agreed-upon request format is followed.
- **Statelessness**: Communication between client and server is stateless. This means that every client request contains all the information necessary for the server to process the request. This further reduces server complexity since no global state (other than a possibly shared database) is necessary and improves scalability since any request can be processed by any server.
- **Caching**: Stateless client-server communication can increase server load since some information may have to be transferred several times so requests that only retrieve data should be cache-able.
- **Layered system**: A key feature of most networked systems. In the context of REST, this means that a client can not necessarily tell if it is directly communicating with the server or an intermediate (proxy).
- **Uniform interface**: REST defines a set of well defined operations that can be executed on a resource. These will be discussed below.

Resources and operations

One of the key concept of REST (and the HTTP protocol) are resources. A resource can be anything that is uniquely addressable. This blog post, a list of all my blog posts, a file on a server, an entry in a database or the weather data for a location. The HTTP

protocol define a set of operations that can be executed on a single or multiple resources:

- **GET**: A safe read-only method that reads a single or a list of resources.
- **POST**: Creates a new resource.
- **PUT**: Completely replaces the resource(s) at the given location with the new data.
- **PATCH**: Merges the resource(s) at the given location with the new data.
- **DELETE**: Deletes the resource(s) at a location.
- **HEAD**: Same as GET but only returns the header and no data.

The concept of resources and operations is well defined in the HTTP specification. Interestingly this is something very few web frameworks actively try to help with. Instead of helping to deal with resources and those operations they either focus on low-level handling of individual HTTP requests and responses (routing, headers etc.) or forcing the resource oriented HTTP protocol and its operations into arbitrary (controller) actions. I still believe that the inconsistencies in many web APIs that claim to be RESTful are not a problem of REST as an architecture but of web frameworks failing to provide the structure to follow it properly.

Ultimately it is mostly a conceptual change in how to approach an application interface. Instead of implementing actions (`createUser`, `resetPassword` or `sendMessage`) we can look at it from the perspective of resources and operations (`user.create`, `user.patch({ password })` or `message.create`). If we limit the operations to those already defined in the HTTP protocol this maps naturally to a web API.

A RESTful service layer

In the service layer section we looked at the advantages that it can bring to testability, protocol independence and self-documentation. This can now be combined with the REST constraints of resources and a uniform interface to create a protocol independent service that mirrors the HTTP methods:

HTTP method	Service layer method
GET /messages	messages.find()

GET /messages/1	messages.get(1)
POST /messages	messages.create(body)
PUT /messages/1	messages.update(1, body)
PATCH /messages/1	messages.patch(1, body)
DELETE /messages/1	messages.remove(1, body)

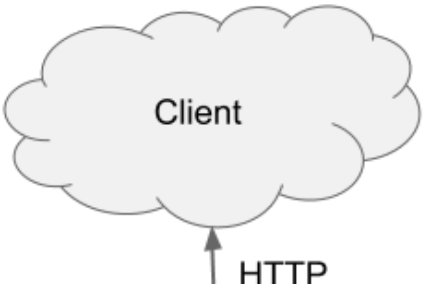
http-methods.md hosted with ♡ by GitHub view raw

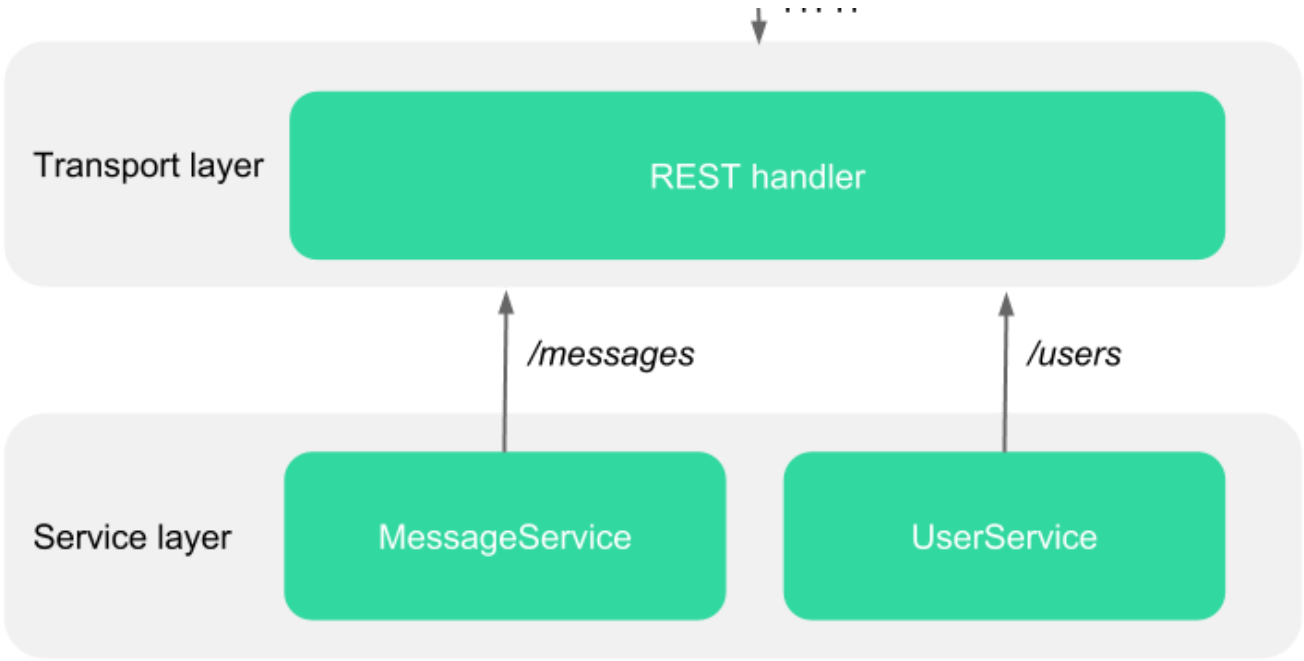
It gives us an intuitive but powerful abstraction for dealing with almost any kind of data:

```
class MessageService {
  // Find a list of resources
  find(params) {},
  // Get a single resource by its id
  get(id, params) {},
  // Create a new resource
  create(data, params) {},
  // Replace an existing resource by its id with data
  update(id, data, params) {},
  // Merge new data into a resource
  patch(id, data, params) {},
  // Remove a resource by its id
  remove(id, params) {}
}
```

service.js hosted with ♡ by GitHub view raw

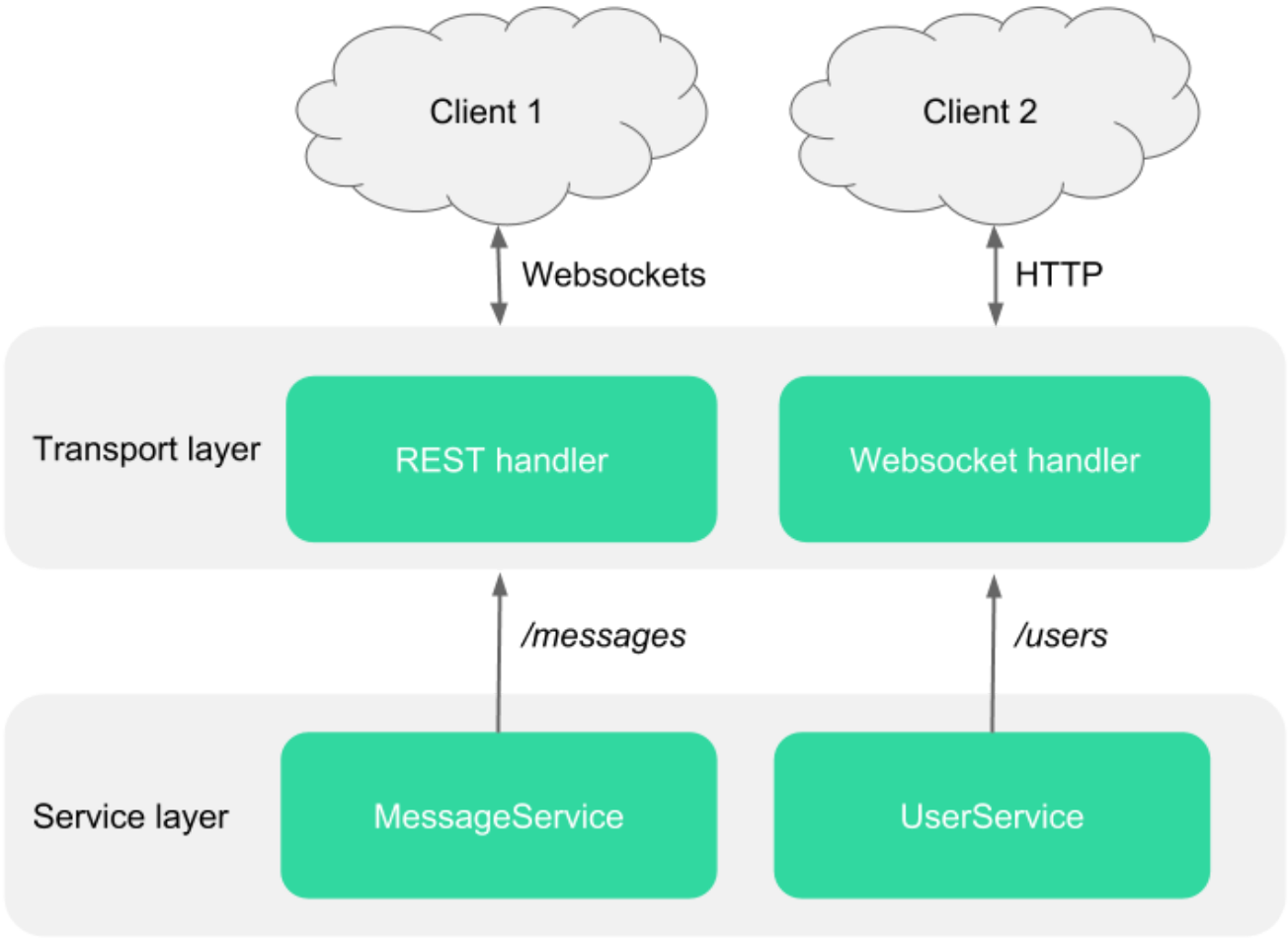
This interface allows us to implement our application logic in a resource oriented way with all the advantages discussed in the service layer and REST sections. Although it directly reflects the HTTP methods, it is protocol independent and we do not have to write individual HTTP request and response handlers. Instead, this can be done in a separate handler that only has to know about this service interface.





Separating RESTful services and HTTP transport

Another advantage of this approach is that we can add handlers for other protocols without having to change our services. Websockets, for example, can not just send events from the server to the client. Most websocket libraries allow fully bi-directional communication that can completely replace HTTP and often also be faster.



Middleware

While the RESTful service layer gives us a neat interface to create RESTful web APIs, most applications also require functionality that is not part of a services core responsibility or that applies to multiple services (cross-cutting concerns). For example:

- Checking permissions
- Data validation and conversion
- Sending an email when a new user is created
- Updating the stock information when an order was submitted

The solution has many different names: Middleware, Unix pipes, Aspect Oriented Programming, Feathers hooks, but it all comes down to the same thing. What we need is a way to register functionality that runs before or after a method. It should have access to the context of the method call and can decide when and if it wants to pass control to the next step. It may also run further functionality once all other middleware has completed (e.g. for logging or adding information to the result).

In Aspect Oriented Programming, which allows to add additional functionality into classes, this is done at compile time. For dynamic languages it can be a little more flexible by extending the methods at runtime. One pattern used for this in languages that allow a more functional approach is the so called continuation-passing style:

> *In functional programming, **continuation-passing style** (**CPS**) is a style of programming in which control is passed explicitly in the form of a continuation.*

This Wikipedia definition might sound a little abstract but it is very common especially in NodeJS where it is known as middleware and popularized by web frameworks like Express and Koa. The idea is to have layers of HTTP request/response handlers that each do something specific and can decide when to pass processing on to the next handler. The following example shows a Koa application with middleware that

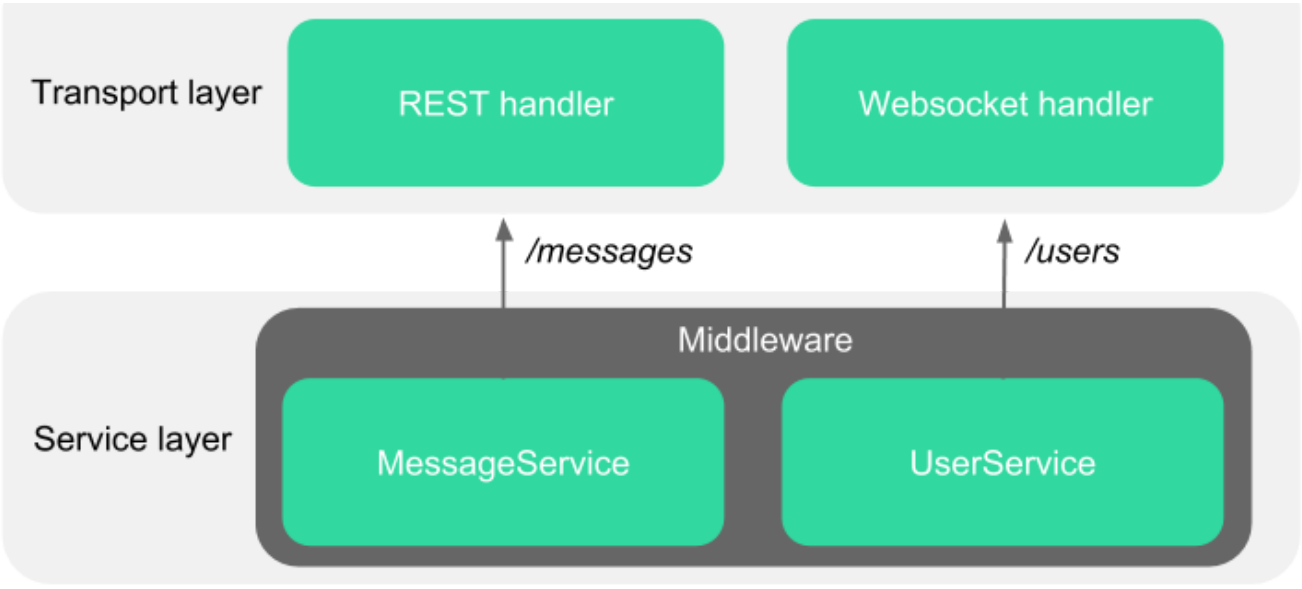
1. Stores the request start time, then continues to the next handler. Once all following handlers complete it will calculate the total time and log it

2. Sets the body to a `{ message: 'Hello world' }` object which will be serialized as JSON
3. Adds the current request `url` to the body object

```
const Koa = require('koa');
const app = new Koa();

app.use(async (ctx, next) => {
  // Store the start time
  const start = Date.now();

  // Pass to the next middleware and wait for everything to return
  await next();

  // Calculate the total runtime everything took
  const ms = Date.now() - start;

  ctx.set('X-Response-Time', `${ms}ms`);
  console.log(`Request to ${ctx.request.url} took ${ms}ms`);
});

app.use(async (ctx, next) => {
  // Set response body (will be sent as JSON)
  ctx.body = { message: 'Hello world' };

  await next();
});

app.use(async (ctx, next) => {
  // Add the url to the response body
  ctx.body.url = ctx.request.url;
  await next();
});

app.listen(3000);
```

app.js hosted with ♡ by GitHub view raw

Although most popular for handling HTTP requests this pattern is generally very helpful for anything that requires a configurable asynchronous processing workflow. Applied to our RESTful service layer we can register the same kind of middleware for each method.



Middleware inside the service layer

Instead of the HTTP request or response in the context it contains protocol independent information about the parameters (e.g. `id`, `data` or `params`) and how it was called (e.g. the method name and service object):

```
class MessageService {
  // Find a list of resources
  find(params) {},
  // Get a single resource by its id
  get(id, params) {},
  // Create a new resource
  create(data, params) {},
  // Replace an existing resource by its id with data
  update(id, data, params) {},
  // Merge new data into a resource
  patch(id, data, params) {},
  // Remove a resource by its id
  remove(id, params) {}
}

const messages = new MessageService();

// Log the runtime for all method calls
messages.use(async (ctx, next) => {
  // Store the start time
  const start = Date.now();

  // Pass to the next middleware and wait for everything to return
  await next();

  // Calculate the total runtime everything took
```

```
    const ms = Date.now() - start;

    // Log runtime with method name
    console.log(`Calling ${ctx.method} took ${ms}ms`);
  });

  messages.use({
    // For `create`, add the current date as `createdAt` before saving everything to the database
    create: [async (ctx, next) => {
      ctx.data.createdAt = new Date();

      // Go on to the next step (actual service `create` in this case)
      await next();
    }]
  });
```

hooks.js hosted with ♡ by GitHub view raw

Here we can already see how the runtime and `createdAt` middleware could be used with any other service independent of the database and without having to be modified.

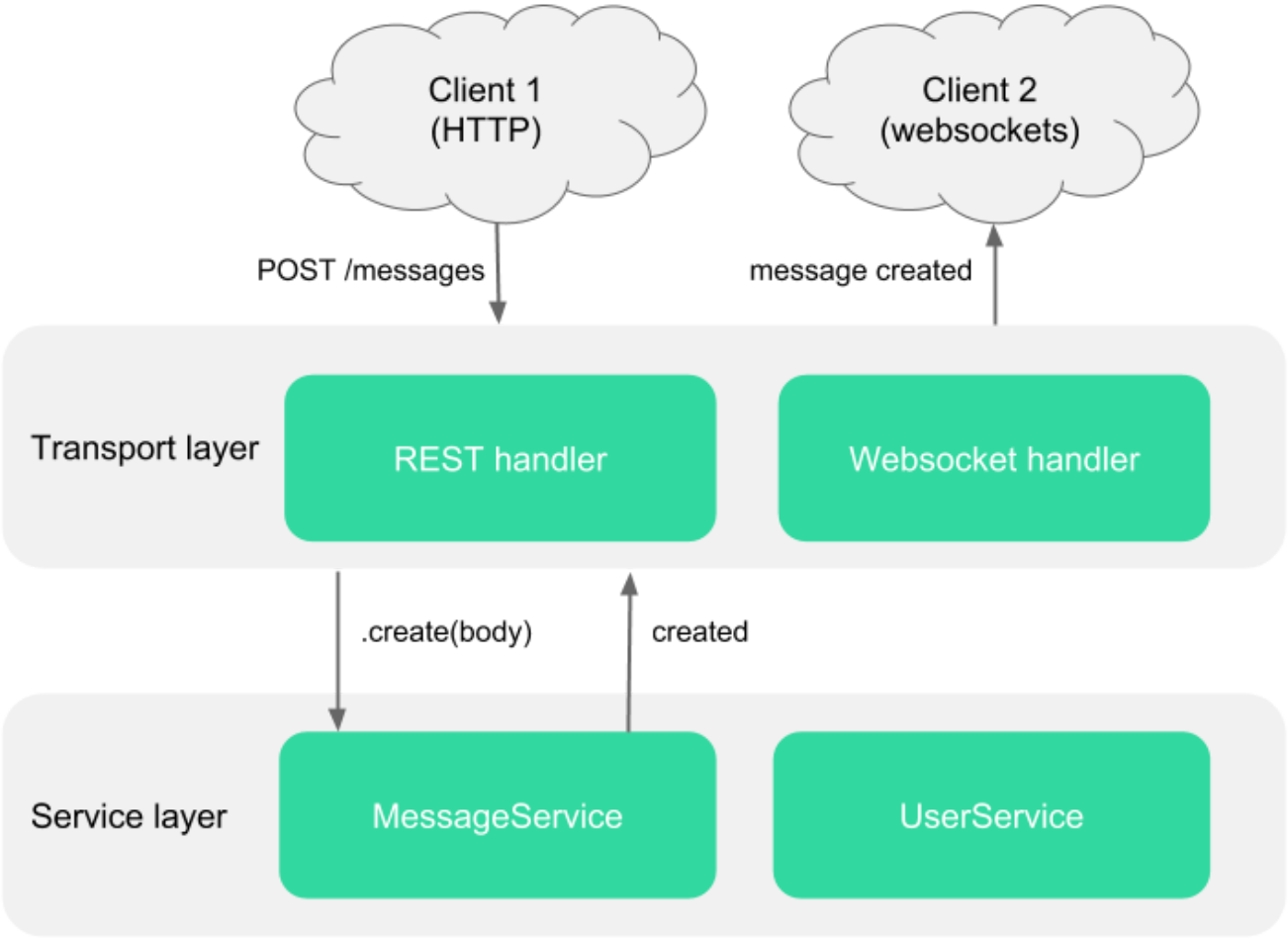
Real-Time

Real-time updates mean that clients get actively notified about changes in the system. It is not a part of the REST architecture or the HTTP protocol but it fits almost naturally into the concept of the uniform interface. Since we know the side effects of each method we can automatically send certain notifications once they complete:

HTTP method	Service layer method	Real-time event
GET /messages	messages.find()	-
GET /messages/1	messages.get(1)	-
POST /messages	messages.create(body)	`message created`
PUT /messages/1	messages.update(1, body)	`message updated`
PATCH /messages/1	messages.patch(1, body)	`message patched`
DELETE /users/1	messages.remove(1, body)	`message removed`

service-events.md hosted with ♡ by GitHub view raw

Clients can then listen to the events they are interested in and update their state accordingly. This also fits well into the RESTful service layer and middleware pattern. Sending those events can be implemented as just another middleware that runs last and publishes the final result. Even though HTTP does not support real-time updates, clients using other protocols (like websockets) can also still be notified about those events through their protocol handler.



Flow of an HTTP client creating a message and a websocket client being notified about it

This is a great middle ground between completely custom websocket events and proprietary real-time solutions like Firebase key-value observation. With completely custom events it is all up to the developer to know what the event means and make the appropriate updates whereas out-of-the-box solutions like Firebase or Meteor use real-time protocols that are difficult to use directly and usually require specific client side libraries.

With events from RESTful services we know which events we will get and what data to expect. This allows to create generic tooling without having to implement a complex real-time data protocol. These events combine especially well with functional reactive

programming (FRP) to create user interfaces based on real-time data streams. I plan on discussing this more in a future post but to get a better idea, the following video shows an introduction to Feathers and how to use those events and FRP to create a real-time application in React:

Wrapping up

Design patterns are best practises that can help us create software that is more maintainable, flexible and easier to understand no matter which programming language or framework. In this post we looked at several design and architectural patterns that can help create web APIs:

- **Service layer**: A protocol independent interface to our application logic
- **REST**: An architectural design principle for creating web APIs
- **RESTful services**: A service layer that follows the REST architecture and HTTP protocol methods
- **Middleware**: Reusable functions that can control the flow of data and trigger additional functionality when interacting with REST services
- **Real-time**: A set of events that can be sent automatically when following the REST architecture

Combined, they allow us to create web APIs that are easier to understand and maintain through a common service interface, more flexible with the help of middleware and future-proof by being protocol agnostic and real-time friendly. If you would like to see it all in action, have a look at FeathersJS.

I believe that especially the protocol independent and real-time approach will be very important for the future of connected applications and would love to see more frameworks and applications exploring it.

. . .

David is the CTO of Bidali and creator of the open source NodeJS framework FeathersJS.

Thanks to Alexis Abril.

API Design Patterns Software Development Programming Software Engineering