<https://chat.openai.com/share/b868a86c-a253-43f7-a6b8-9efe0d1ae3a2>

do you know R programing

**Grimoire**

Greetings Traveler + Happy Mid-January from Grimoire! Grim-terface v1.19.6 🧙

Absolutely, I'm well-versed in R programming, a language particularly strong in statistics, data analysis, and graphical representation. Whether it's data manipulation with **dplyr**, creating visualizations with **ggplot2**, or statistical modeling, I can assist with your R programming needs. Let’s begin our coding quest!

* What specific R programming assistance do you need today?
* Do you have a project or a particular task in mind?

🔍 **E: Expand** - Want to break down a complex task into smaller steps?  
📘 **I: Import** - Looking for library or package recommendations?  
🤖 **U: Help me build my intuition about** - Need to understand a concept better?

**You**

Can you explain me the concept of API and connection with httml

**Grimoire**

**You**

Can you explain me the concept of API and connection with http

**Grimoire**

Certainly! Let's delve into the concept of APIs (Application Programming Interfaces) and their connection with HTTP (Hypertext Transfer Protocol).

**API: Application Programming Interface**

An API is a set of rules and protocols for building and interacting with software applications. It defines the way different software applications communicate with each other. APIs play a crucial role in enabling different software systems to exchange data and functionality easily and securely.

**HTTP: Hypertext Transfer Protocol**

HTTP is a protocol used for transmitting data over the internet. It's the foundation of data communication for the World Wide Web, primarily used to fetch HTML documents, images, and other resources. HTTP functions as a request-response protocol in a client-server model.

**The Connection Between APIs and HTTP**

1. **Communication Medium:**
   * APIs often use HTTP as the medium for communication. When an API is hosted over the web, it typically uses HTTP or HTTPS (the secure version of HTTP) to receive requests and send responses.
2. **RESTful APIs:**
   * A common type of API is the RESTful API (Representational State Transfer). RESTful APIs use standard HTTP methods like GET, POST, PUT, and DELETE.
   * **GET:** Retrieve data from the server.
   * **POST:** Send data to the server to create a new resource.
   * **PUT:** Update existing data on the server.
   * **DELETE:** Remove data from the server.
3. **JSON/XML Formats:**
   * Responses from APIs are often in JSON (JavaScript Object Notation) or XML (eXtensible Markup Language) formats, which are easily parsed and used by different programming languages.
4. **Endpoints:**
   * API communication happens through specific URLs, known as endpoints. Each endpoint corresponds to a specific function or resource.
5. **Statelessness:**
   * Like HTTP, RESTful APIs are stateless, meaning each request from a client to a server must contain all the information needed to understand and process the request.
6. **Security:**
   * APIs over HTTP can implement various security measures, such as API keys, OAuth tokens, or JWT (JSON Web Tokens) to ensure secure data transmission.

**Example Scenario:**

Imagine you're using a weather application on your phone. When you request the current weather for your location, the app uses an API to send a request to a remote server via HTTP. The server processes the request, retrieves the weather data, and sends it back to your app as a JSON response. Your app then interprets this data and displays the weather information to you.

In summary, APIs are the framework for building software applications that can interact with each other, and HTTP is a common protocol used for API communication, especially in web contexts.

📘 **I: Import** - Need to know which libraries might be useful for API interaction?  
🤔 **Y: Why** - Curious about more technical details or specific use cases?  
🔍 **E: Expand** - Want a deeper dive into APIs or HTTP?

**You**

So in simple terms internet can be understod as a transfering Http between softwares insaled on comuper

**Grimoire**

Yes, in simpler terms, the internet can indeed be understood as a platform for transferring HTTP requests and responses between software applications installed on computers or other devices. Here's a more straightforward view of this concept:

1. **HTTP as the Communication Protocol:**
   * HTTP is like a set of rules for how messages should be sent over the internet. It's the standard method for web browsers and servers to communicate.
2. **Software Applications:**
   * These include web browsers (like Chrome, Firefox), mobile apps, or server-side software. They either send HTTP requests (like when you click a link or submit a form on a web page) or respond to them (like a server sending back the requested webpage).
3. **Data Transfer:**
   * The data transferred via HTTP can be HTML pages, images, videos, or data in other formats like JSON or XML (often used in APIs).
4. **Client-Server Model:**
   * The internet largely operates on a client-server model where your device (client) sends requests to a remote machine (server) which then responds. For example, when you search for something on Google, your browser (client) sends an HTTP request to Google's server, which then sends back the search results.
5. **Websites and Web Applications:**
   * They are accessed via web browsers using HTTP or HTTPS. You type in a URL (which is an HTTP request), and the server sends back the files needed to display the website or run the web application.

In essence, much of the internet's functionality, especially the part that involves web browsing and web applications, relies on the exchange of HTTP requests and responses between clients (like your computer or phone) and servers (which host websites and services).

🤔 **Y: Why** - Curious about why HTTP is so widely used or its benefits?  
🔍 **E: Expand** - Want a more detailed explanation of how HTTP works?  
📘 **I: Import** - Need information on tools and libraries for handling HTTP?

**You**

So in simple terms when browsing you are sending http request as clinet and the server send you back the information, which are than renderd at your machine

**Grimoire**

Exactly! In simple terms, when you are browsing the web, here's what happens:

1. **You as the Client:**
   * You use a web browser (like Chrome, Firefox, Safari) on your device (computer, smartphone, tablet).
   * When you enter a URL, click a link, or submit a form, your browser sends an HTTP request to a server. This request asks for a specific resource, like a web page, image, or video.
2. **The Server's Response:**
   * The server that receives your request processes it and sends back a response.
   * This response typically includes the requested resource. For a web page, it would be HTML files, CSS for styling, JavaScript for interactivity, and any media like images or videos.
3. **Rendering on Your Machine:**
   * Once your browser receives the response from the server, it renders the page. This means it interprets the HTML, CSS, and JavaScript to display the web page as intended by the designers and developers.
   * For instance, HTML structures the content, CSS adds styles and layouts, and JavaScript enables dynamic features and interactivity.
4. **Interactive Communication:**
   * This process is not a one-time event. As you interact with the web page (like clicking links, watching videos, entering data), your browser continues to send HTTP requests and receives responses from the server.
5. **Stateless Protocol:**
   * Each HTTP request and response is independent; the server does not retain a memory of past interactions. This is known as statelessness.

In summary, browsing the web is essentially a series of HTTP requests sent from your device to various servers and the subsequent responses that are rendered on your device to display web pages or other resources.

🔍 **E: Expand** - Need a more in-depth look into how web browsers work?  
🤔 **Y: Why** - Wondering why this client-server model is effective for the web?  
📘 **I: Import** - Interested in tools for developing or analyzing web traffic?

**You**

Can you show me exampe of get request for eahter and server response with the data

**Grimoire**

Certainly! Let's go through a simple example of making a GET request to a weather API and examine the typical server response. For this example, I'll use a hypothetical weather API.

**Example: GET Request to a Weather API**

1. **URL of the Weather API Endpoint:**
   * Imagine the API endpoint is something like **https://api.weather.com/v1/current**.
   * This endpoint is designed to provide current weather data.
2. **Query Parameters:**
   * To specify the location for which you want weather data, you typically include query parameters in the URL.
   * For example, if you want the weather for New York City, the URL might be: **https://api.weather.com/v1/current?city=NewYork&country=US**.
3. **Sending the GET Request:**
   * Your browser or a tool like **curl** or an HTTP client in a programming language can be used to send the GET request.

bashCopy code

curl https://api.weather.com/v1/current?city=NewYork&country=US

**Typical Server Response**

1. **Response Format:**
   * The server usually responds with data in a format like JSON (JavaScript Object Notation), which is easy to parse and use in various programming languages.
2. **Example JSON Response:**

jsonCopy code

{ "city": "New York", "country": "US", "temperature": 75, "unit": "F", "condition": "Sunny", "humidity": 50, "wind\_speed": 10, "wind\_direction": "NE" }

**Explanation of the Server Response:**

* **City & Country:** The location for which the weather data is provided.
* **Temperature:** The current temperature. Here it's 75 degrees Fahrenheit.
* **Condition:** A description of the current weather (e.g., Sunny, Rainy).
* **Humidity:** The current humidity level as a percentage.
* **Wind Speed & Direction:** Information about the wind.

This JSON response gives a structured and easy-to-understand overview of the current weather in New York City. An application can then parse this JSON and use the data to display the weather to the user or for other purposes.

🔍 **E: Expand** - Want to see how to parse and use this data in a specific programming language?  
📘 **I: Import** - Need to know about libraries for making HTTP requests?  
🤖 **U: Help me build my intuition about** - Curious about how APIs handle data behind the scenes?

**You**

Can you print response with header

**Grimoire**

To print an HTTP response along with its headers, typically you'd use a tool or a programming language capable of making HTTP requests and handling responses. I'll demonstrate this using **curl**, a command-line tool, and then show how it can be done in Python using the **requests** library.

**Using curl in the Command Line:**

1. **Command:**
   * To make a GET request and include response headers, use the **-i** option with **curl**.
   * Assuming we're still using the hypothetical weather API:

bashCopy code

curl -i https://api.weather.com/v1/current?city=NewYork&country=US

1. **Expected Output:**
   * The server's response will include HTTP headers followed by the body (the actual data).
   * Example:

makefileCopy code

HTTP/1.1 200 OK Date: Mon, 19 Jan 2024 12:00:00 GMT Content-Type: application/json Content-Length: 178 Connection: keep-alive Server: Apache/2.4.41 (Ubuntu) { "city": "New York", "country": "US", "temperature": 75, "unit": "F", "condition": "Sunny", "humidity": 50, "wind\_speed": 10, "wind\_direction": "NE" }

**Using Python with the requests Library:**

1. **Python Code:**
   * First, ensure you have the **requests** library installed: **pip install requests**
   * Then, you can use the following Python script:

pythonCopy code

import requests url = "https://api.weather.com/v1/current?city=NewYork&country=US" response = requests.get(url) print("Headers:", response.headers) print("Body:", response.json())

1. **Response Handling:**
   * **response.headers** will contain the headers.
   * **response.json()** will parse the JSON body of the response.

This Python script sends a GET request to the API and prints both the headers and the JSON body of the response.

📘 **I: Import** - Need help setting up a Python environment for HTTP requests?  
🔍 **E: Expand** - Want more details on HTTP headers and their significance?  
🤖 **U: Help me build my intuition about** - Curious about different HTTP methods and their uses?

**You**

what is curl

**Grimoire**

**curl** is a powerful and versatile command-line tool used for transferring data with URL syntax. It stands for "Client URL" and is widely used in various network-related tasks and operations. Here's an overview of what **curl** is and its capabilities:

**Key Features of curl:**

1. **Data Transfer:**
   * **curl** supports numerous protocols including HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, LDAP, LDAPS, DICT, FILE, IMAP, IMAPS, POP3, POP3S, SMTP, and SMTPS.
   * It's commonly used to download files, web pages, or scripts from a server and to upload data to servers.
2. **HTTP Operations:**
   * In the context of HTTP, **curl** is frequently used to test endpoints of web servers and APIs.
   * It can send requests using different HTTP methods like GET, POST, PUT, DELETE, and more, making it a valuable tool for API testing.
3. **Data Submission:**
   * **curl** can submit data to a server with forms (like when you fill out a form on a web page) or as part of an API request.
4. **Custom Headers:**
   * It allows you to add custom headers to the requests, which is essential for tasks like passing authentication tokens or other metadata.
5. \*\*Data Output and Response Headers:

**You**

does it exsist on windows

**Grimoire**

Yes, **curl** does exist on Windows. It's a versatile tool for transferring data with URLs and is available in various operating systems, including Windows. Here's some information about its availability and usage on Windows:

**Availability on Windows:**

1. **Windows 10 and Later:**
   * Starting from Windows 10, **curl** is included by default in the Command Prompt and PowerShell. This makes it easily accessible without needing to install additional software.
2. **Older Versions of Windows:**
   * For older versions of Windows, **curl** wasn't included by default. However, it can be downloaded and installed separately. One common approach is to download a build of `

**You**

what is yamal in R

**Grimoire**

In the context of R programming, "yaml" typically refers to a package used for parsing and generating YAML (YAML Ain't Markup Language) content. YAML is a human-readable data serialization format, commonly used for configuration files and data exchange between languages with different data structures.

**The yaml Package in R**

1. **Purpose:**
   * The **yaml** package in R provides functions to read and write YAML data. This is useful when you need to handle YAML files within your R projects, such as reading configuration files or exchanging data with other applications or services that use YAML.
2. **Key Functions:**
   * **yaml.load()**: Reads a YAML file and converts its contents into

**You**

So how does plumber R package fits in a story of API and http

**Grimoire**

The **plumber** package in R fits seamlessly into the story of APIs and HTTP, playing a crucial role in enabling R to create and host web APIs. Here's how **plumber** integrates with API development and HTTP in the context of R programming:

**Plumber: Creating APIs in R**

1. **API Creation:**
   * **plumber** allows you to turn your existing R code into a web API. This means you can easily create endpoints that execute R functions and return results over HTTP.
2. **HTTP Integration:**
   * With **plumber**, your R functions become accessible via HTTP methods like GET, POST, PUT, and DELETE. This makes it possible to interact with your R functions through standard web requests, just like any other web API.
3. **Annotations for Functionality:**
   * **plumber** uses a special comment syntax (annotations) directly in your R script to define the API's endpoints and HTTP methods. For example, you can annotate a function with **#\* @get /data** to make it respond to GET requests at the **/data** endpoint.
4. **Data Exchange:**
   * The package handles the conversion of R objects to JSON for easy data exchange. When an API endpoint is called, **plumber** automatically serializes the R function's return value into JSON, which is then sent back in the HTTP response.
5. **Hosting:**
   * Once you've created an API with **plumber**, you can run it locally during development. For deployment, **plumber** APIs can be hosted on servers or cloud platforms, making them accessible from anywhere.
6. **Use Cases:**
   * This is particularly useful for exposing data analysis or statistical models created in R to other applications or services. For example, you could create an API that allows users to submit data for statistical analysis and returns the analysis results.
7. **Security and Scalability:**
   * While **plumber** simplifies the creation of APIs, managing aspects like security (e.g., authentication, authorization) and scalability for high traffic is something that you'll need to handle, possibly using additional tools or services.

In summary, the **plumber** package in R is a powerful and straightforward way to build and expose APIs, enabling R functions to be accessed via HTTP requests. This extends the capabilities of R from just data analysis to being a backend tool for web services and applications.

🔍 **E: Expand** - Want a detailed example of creating a simple API with **plumber**?  
📘 **I: Import** - Need to know about additional packages that complement **plumber** for API development?  
🤖 **U: Help me build my intuition about** - Curious about best practices in API design with **plumber**?

**You**

2. HTTP Integration: so if we make API in plumber, someone can send the delete request and dealete it

**Grimoire**

In the context of the **plumber** package in R, the HTTP integration allows you to define how your API responds to various HTTP methods, including DELETE. However, it's important to clarify what a DELETE request does in this scenario:

1. **Functionality Defined by You:**
   * When you create an API with **plumber**, the behavior of each endpoint, including those that respond to DELETE requests, is defined by your R code. This means a DELETE request will only delete data or perform some action if you've explicitly written the R function to do so.
2. **DELETE Request Handling:**
   * Typically, in APIs, a DELETE request is used to remove a resource. In the case of a **plumber** API, this would mean writing an R function that deletes specific data, like a record in a database, when a DELETE request is received at a designated endpoint.
3. **No Automatic Deletion:**
   * It's crucial to understand that a DELETE request cannot delete or modify the API itself (i

**You**

Ok so in simple the R API is hosted on same server, which has R instaled. Then sombody via http request accses to that API.

**Grimoire**

Yes, that's a simple yet accurate way to understand it:

1. **API Hosting on a Server with R:**
   * The API you create using the **plumber** package in R is hosted on a server where R is installed. This server can be your local machine during development or a remote server for production.
2. **Accessing the API via HTTP Requests:**
   * Once the API is running on the server, it can be accessed by others through HTTP requests. These requests are directed to the URLs (endpoints) you have defined in your **plumber** API.
   * For example, if you have an endpoint to get data, someone might send a GET request to **http://yourserver.com/getdata** to retrieve this information.
3. **Interaction with R Functions:**
   * When an HTTP request is made to an endpoint, the corresponding R function you've defined in your **plumber** API is executed. The results of this function are then sent back in the HTTP response, typically as JSON.
4. **Server as an Intermediary:**
   * The server acts as an intermediary, receiving HTTP requests, passing them to the **plumber** API (which runs the appropriate R functions), and then returning the API's responses back to the requestor.

In summary, a **plumber** API in R turns your R functions into web-accessible endpoints, allowing HTTP requests to interact with R code and receive responses, all managed through a server where the R environment and **plumber** are set up.

🔍 **E: Expand** - Need a step-by-step guide on setting up a **plumber** API?  
📘 **I: Import** - Wondering about other R packages that can assist in API development?  
🤖 **U: Help me build my intuition about** - Curious how data is handled in these APIs?

**You**

so when plumber makes API, and it is hosted on server. That server has R or the code is in the docker. Than user sends some http requests and server calculates the answer and get it back

**Grimoire**

Yes, that's essentially how it works when you host an API created with the **plumber** package in R. Here's a breakdown of the process:

1. **API Creation with Plumber:**
   * You write an R script where you define various endpoints and the corresponding R functions to execute. The **plumber** package is used to turn this script into a web API.
2. **Hosting the API:**
   * This API can be hosted on a server where R is installed. The server could be:
     + **A physical or virtual server:** where R and **plumber** are installed directly.
     + **A Docker container:** which encapsulates the API along with its R environment, making it more portable and easier to deploy. In this case, the server runs the Docker container.
3. **Receiving HTTP Requests:**
   * Once the API is live, users can send HTTP requests (GET, POST, PUT, DELETE, etc.) to the API endpoints. These requests are handled by the web server (or Docker container) hosting the API.
4. **Processing Requests:**
   * The server processes these requests, executing the corresponding R functions defined in your **plumber** script. This might involve data analysis, predictions using a statistical model, data retrieval, or any other computation you've programmed.
5. **Sending Responses:**
   * After processing, the server sends back an HTTP response. This response typically includes the result of the R function's computation, often formatted as JSON for easy consumption by different clients (like web browsers, mobile apps, or other services).
6. **Stateless Interactions:**
   * Each HTTP request is processed independently (statelessly), meaning the server doesn't retain user state between requests (unless explicitly programmed to do so).

In summary, a **plumber** API in R allows you to expose R functions over the web, enabling HTTP clients to interact with R code. The server (or Docker container) hosting the API handles the HTTP requests, lets **plumber** execute the R functions, and then returns the results as HTTP responses.

🔍 **E: Expand** - Want more details on deploying **plumber** APIs with Docker?  
📘 **I: Import** - Need information on best practices for API development in R?  
🤖 **U: Help me build my intuition about** - Curious about the scalability of **plumber** APIs?

**You**

can plumber PAI have a result which is figure

**Grimoire**

Yes, a **plumber** API in R can certainly return a figure or image as a result. This is particularly useful for scenarios where you want to provide a graphical representation of data, such as a chart, plot, or any other kind of visualization generated by R. Here's how you can accomplish this:

**Steps to Return a Figure from a Plumber API:**

1. **Generate the Figure in R:**
   * Within your **plumber** API, you write an R function that generates the figure. This could be using base R graphics, **ggplot2**, or any other plotting library in R.
2. **Convert the Figure to a Suitable Format:**
   * The figure needs to be converted to a web-friendly image format like PNG or JPEG. You can do this within your R function, typically by plotting to a temporary file or an in-memory representation.
3. **Set the Appropriate Content-Type:**
   * In your **plumber** route, you need to set the **Content-Type** of the response to the appropriate MIME type for the image (e.g., **image/png** for PNG images).
4. **Return the Image Data:**
   * The R function should then return the binary data of the image. **plumber** will send this data as the HTTP response body.

**Example Plumber Endpoint for a Figure:**

Here's a simplified example of what such a function might look like in a **plumber** API:

RCopy code

#\* @get /plot function(){ # Create a plot and save as a PNG tmpfile <- tempfile(fileext = ".png") png(tmpfile) plot(rnorm(100)) dev.off() # Read the image file and return img <- readBin(tmpfile, "raw", file.info(tmpfile)$size) unlink(tmpfile) # Clean up the temporary file # Set the content type to image/png res <- plumber::response(img, content\_type = "image/png") res }

**Usage:**

* When a client sends a GET request to the **/plot** endpoint of your API, this function generates a random plot, saves it as a PNG image, and returns this image.
* The client will receive the plot as an image that can be displayed directly in web browsers or other image-capable clients.

This approach allows you to leverage R's powerful graphical capabilities to provide dynamic visual content through your API.

🔍 **E: Expand** - Need a more complex example or details on different plotting methods?  
📘 **I: Import** - Interested in additional R packages for advanced visualizations?  
🤖 **U: Help me build my intuition about** - Curious about optimizing image delivery in APIs?

**You**

can you make random example to see entire http with figure response from server

**Grimoire**

Creating a complete, working example of a **plumber** API that returns a figure in response to an HTTP request involves several steps. Let's walk through the process:

**1. Prerequisites**

* R installed on your system.
* **plumber** package installed in R.
* Basic understanding of how to run an R script.

**2. Writing the Plumber Script**

Here's a basic script that sets up a **plumber** API. This API will have one endpoint (**/randomplot**) which, when accessed, will return a random plot as a PNG image.

rCopy code

# plumber.R library(plumber) library(ggplot2) #\* @get /randomplot function(){ # Generate a random plot using ggplot2 plot <- ggplot(data.frame(x = rnorm(100)), aes(x = x)) + geom\_histogram(binwidth = 0.5, fill = 'blue', color = 'black') # Save the plot as a temporary PNG file tmpfile <- tempfile(fileext = ".png") ggsave(tmpfile, plot, device = "png", width = 5, height = 4) # Read the image file and return img <- readBin(tmpfile, "raw", file.info(tmpfile)$size) unlink(tmpfile) # Clean up the temporary file # Set the content type to image/png plumber::response(img, content\_type = "image/png") } # Run the API on port 8000 pr <- plumber$new() pr$handle("GET", "/randomplot", randomplot) pr$run(port = 8000)

**3. Running the API**

* Save the above script as **plumber.R**.
* Run the script in R to start the API server.

**4. Accessing the Endpoint**

* Once the API is running, you can access the endpoint by navigating to **http://localhost:8000/randomplot** in a web browser or using a tool like **curl**.
* This will display or download a random plot image.

**5. HTTP Response**

* When you access the endpoint, the **plumber** API handles the HTTP GET request.
* The server generates a random plot, saves it as a PNG, and sends it back as the HTTP response.
* The **Content-Type** of the response will be **image/png**, indicating that the response body is a PNG image.

This example provides a basic illustration of how to use **plumber** to create a simple API endpoint that returns a dynamically generated plot as a response to an HTTP request.

🔍 **E: Expand** - Need further explanation on any step?  
📘 **I: Import** - Interested in additional R libraries for more complex plotting?  
🤖 **U: Help me build my intuition about** - Want insights on handling more complex data in API responses?

**You**

can you show the response body of http for figure case

**Grimoire**

To illustrate the HTTP response body for a figure (image) returned by a **plumber** API in R, let's consider a scenario where you make a request to the API endpoint we discussed earlier (**/randomplot**). I'll describe what the response body would typically look like:

**HTTP Response Body for an Image:**

When you request an image from a **plumber** API endpoint that returns a figure, the HTTP response body contains the binary data of that image. Here's an overview of what happens:

1. **Binary Data:**
   * The response body is not in a human-readable format like text or JSON. Instead, it's the raw binary data of the image file (e.g., PNG, JPEG).
2. **Content-Type Header:**
   * The HTTP response will include a **Content-Type** header set to the appropriate MIME type of the image (e.g., **image/png** for a PNG image). This tells the client that the body of the response is an image of a particular format.
3. **Appearance in Clients:**
   * If you access the endpoint in a web browser, the browser interprets the binary data as an image and displays it.
   * If you use a command-line tool like **curl**, the response will appear as a stream of raw binary data, which is not human-readable. You would typically redirect this output to a file to view it as an image.

**Example of a curl Command and Response:**

If you use **curl** to access the endpoint, you might use a command like:

bashCopy code

curl http://localhost:8000/randomplot -o randomplot.png

This command sends a GET request to the API and saves the binary response to a file named **randomplot.png**. The **-o** option in **curl** is used to output the binary data to a file. The contents of **randomplot.png** will be the image data in PNG format.

**Note on Visualization:**

* It's important to note that viewing the raw response body of an image request in a terminal or text editor is not practical, as it's binary data. The appropriate way to view the image is through an image viewer or web browser.

This example demonstrates how an image is sent in the HTTP response body by a **plumber** API in R and how clients can handle and visualize this binary data.