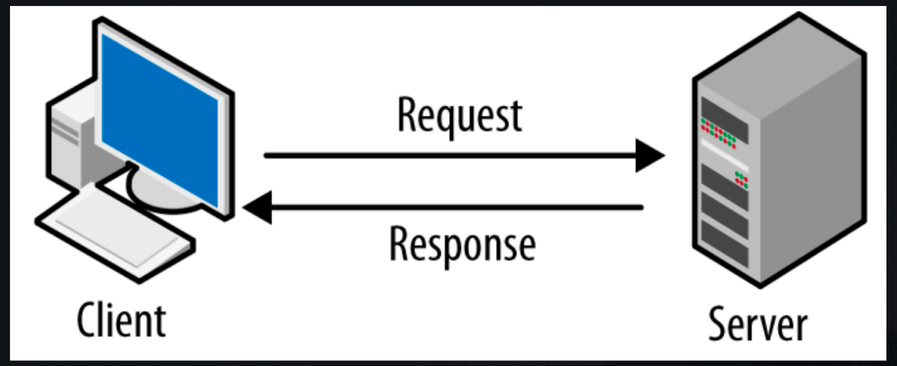
# Making API Requests

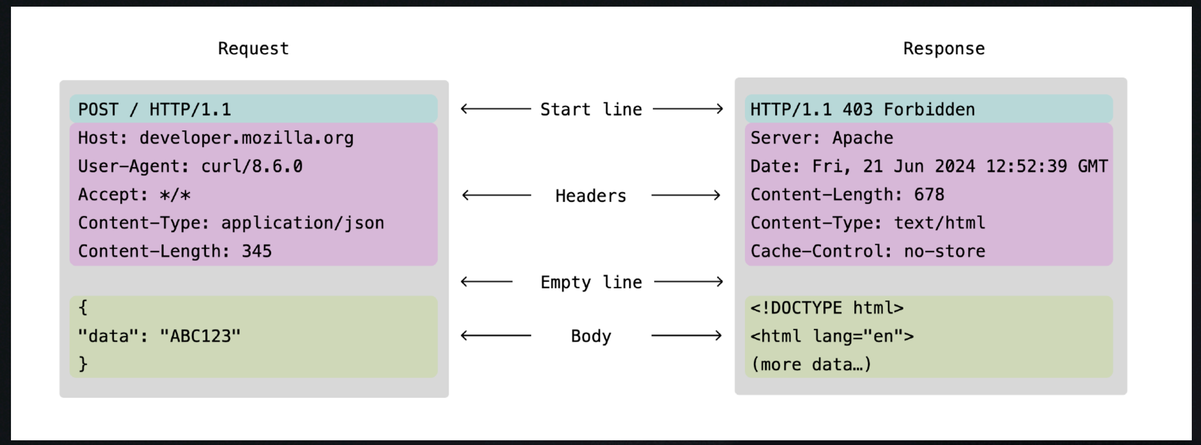
## Exploring the Client-Server Relationship

# Client 🤝 Server




- Clients and servers are *roles* that computers can play with each other
- Client must initiate by sending a request
- Server is expected to send one (and only one) response
- HTTP (HyperText Transfer Protocol) is a set of rules governing these interactions

# HTTP




- An HTTP request has a method and a path
  - e.g. GET and /
  - Sometimes, a body with data
- An HTTP response has a status code
  - Sometimes, a body with data

# HTTP Request Methods

| Method | Meaning |
| --- | --- |
| GET /puppies | Give me all the puppies  |
| GET /puppies/2 | Give me the puppy with id 2  |
| POST /puppies | Here's a new puppy  |
| DELETE /puppies/4 | Remove puppy with id 4  |
| PATCH /puppies/7 | Update puppy with id 7  |
| PUT /puppies/9 | Replace all data about puppy 9 with this data  |

NOTE: These are just some of the most commonly used HTTP methods

# HTTP Request Methods

## Stripe

### Charges

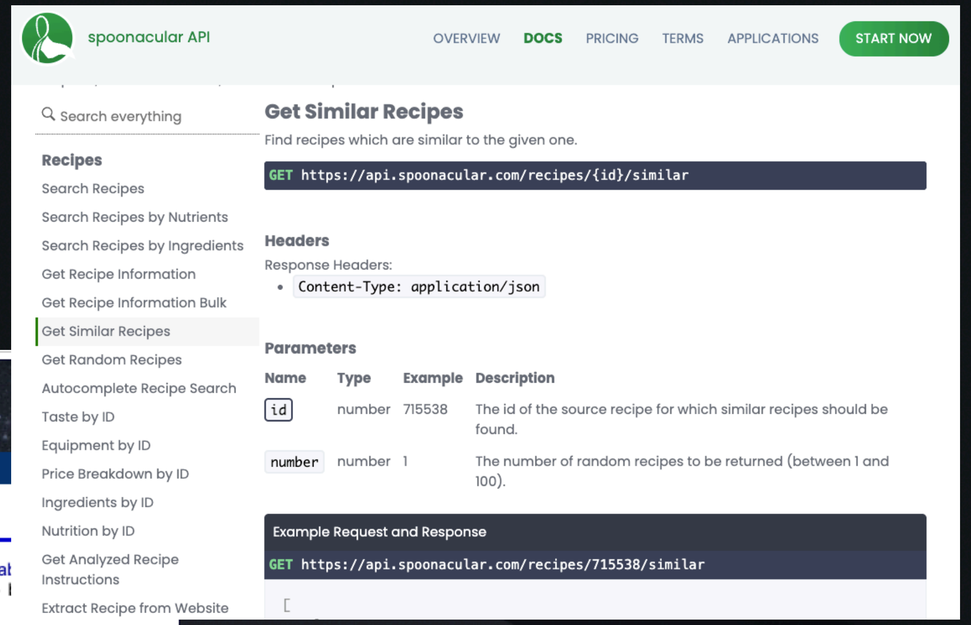
The `Charge` object represents a single attempt to move money into your Stripe account. PaymentIntent confirmation is the most common way to create Charges, but transferring money to a different Stripe account through Connect also creates Charges. Some legacy payment flows create Charges directly, which is not recommended for new integrations.

Was this section helpful? Yes No

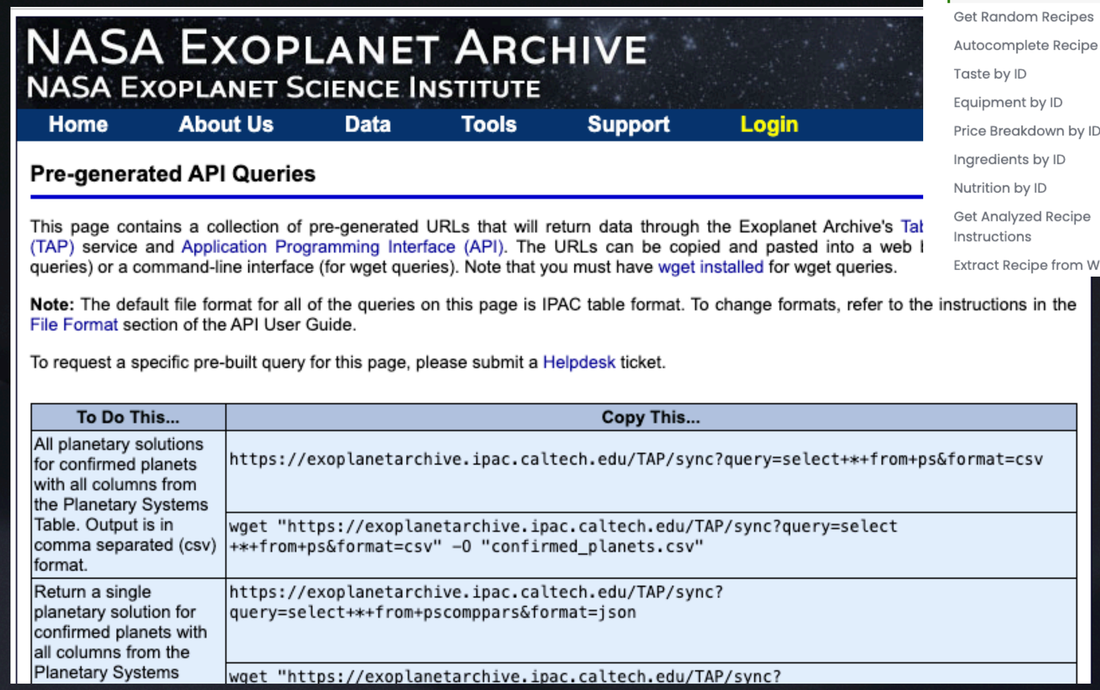
**ENDPOINTS**

```
POST /v1/charges
POST /v1/charges/:id
 GET /v1/charges/:id
 GET /v1/charges
POST /v1/charges/:id/capture
 GET /v1/charges/search
```

## Spoonacular




## NASA



# Making Requests From Browser JS


- fetch() is built-in to browsers, like document
- Allows us to requests data from a server
- Returns a Promise

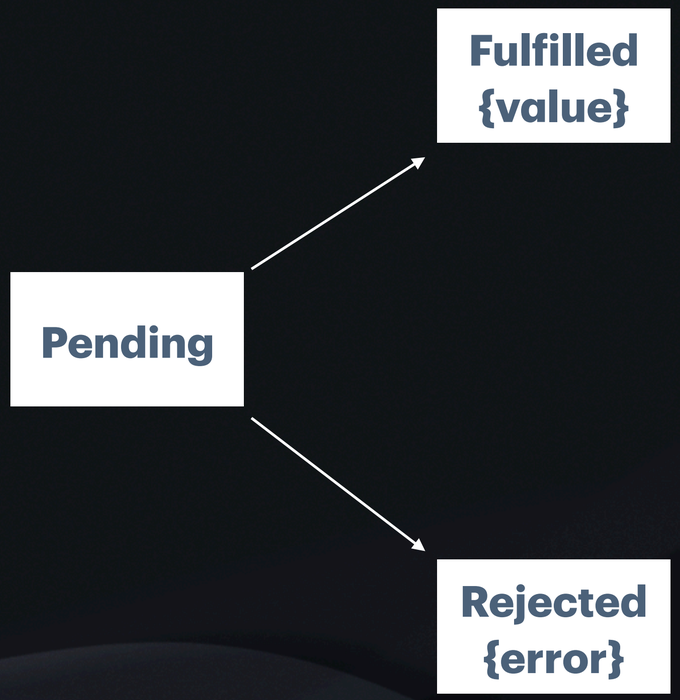
The `fetch()` function returns a `Promise` which is fulfilled with a `Response` object representing the server's response. You can then check the request status and extract the body of the response in various formats, including text and JSON, by calling the appropriate method on the response.

Here's a minimal function that uses `fetch()` to retrieve some JSON data from a server:

JS

```
async function getData() {
  const url = "https://example.org/products.json";
  try {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error(`Response status: ${response.status}`);
    }

    const json = await response.json();
    console.log(json);
  } catch (error) {
    console.error(error.message);
  }
}
```

# Promises




- “A Promise represents the eventual completion of an asynchronous operation and its resulting value” - MDN
- Promises always start in a Pending state
- If the operation is successful, the promise will be “fulfilled” and have a value attached
- If the operation is unsuccessful, the promise will be “rejected” and have an error attached
- Once the promise resolves to fulfilled or rejected, it cannot change back. It’s done!

# Promises (.then)

```
const promise = fetch("https://jsonplaceholder.typicode.com/posts");
console.log(promise); // Promise {<pending>}
const rawData = promise.then((response) => {
  console.log(response);
  return response.json();
});
const jsonData = rawData.then((json) => {
  console.log(json);
  return json;
});
console.log(jsonData); // Promise {<pending>}
```

# Promises (await)

```
async function fetchData() {
  const response = await fetch("https://jsonplaceholder.typicode.com/posts");
  const data = await response.json();
  console.log(data);
  return data;
}
fetchData();
```

# Promises (await)

```
async function bakeBread() {
  const flour = await getFlour();
  const yeast = await getYeast();
  const water = await getWater();
  const salt = await getSalt();
  const knead = await kneadBread(flour, yeast, water, salt);
  const bake = await bakeBread(knead);
  return bake;
}
bakeBread();
```

- await pauses execution until the promise is fulfilled
  - The next line will not run until the promise is fulfilled
- await must be within an async function
  - Sometimes, we create an async function and then immediately invoke it
- async functions *always* return a Promise

# Back to React

```
const url = "https://myblog.com/posts";

const App = () => {
  const [data, setData] = useState([]);

  const fetchData = async () => {
    const response = await fetch(url);
    const data = await response.json();
    setData(data);
  };
  fetchData();

  return (
    <div>
      {data.map((item) => (
        <div key={item.id}>{item.title}</div>
      ))}
    </div>
  );
};
```

- When the React component first renders, it hasn't made the request yet
  - The first time the React component renders is called "mounting" 🏇
- We should initialize the state as something empty, or somehow represent a Loading state
- Then, when the component finishes setting data, it re-renders
- Which triggers another fetch()...
- Which triggers another re-render

# React useEffect

```
const url = "https://myblog.com/posts";

const App = () => {
  const [data, setData] = useState([]);

  const fetchData = async () => {
    const response = await fetch(url);
    const data = await response.json();
    setData(data);
  };
  useEffect(() => {
    fetchData();
  }, [])

  return (
    <div>
      {data.map((item) => (
        <div key={item.id}>{item.title}</div>
      ))}
    </div>
  );
};
```

dependencies

- useEffect is a hook, similar to useState
  - Don’t forget to import it at top of file:
  - `import { useState, useEffect } from ‘react’`
- Unlike useState, useEffect doesn’t return anything
- useEffect takes a callback function
- It runs that callback function after the component mounts
- And then again whenever one of the dependencies changes
  - [] dependencies means “don’t run it again”

# React useEffect

```
const App = () => {
  const [data, setData] = useState([]);
  const [counter, setCounter] = useState(0);
  const triggerEffect = () => {
    setCounter(counter + 1);
  };

  const fetchData = async () => {
    const response = await fetch(url);
    const data = await response.json();
    setData(data);
  };
  useEffect(() => {
    fetchData();
  }, [counter]);

  return (
    <div>
      <button onClick={triggerEffect}>Fetch Data</button>
      {data.map((item) => (
        <div key={item.id}>{item.title}</div>
      ))}
    </div>
  );
};
```

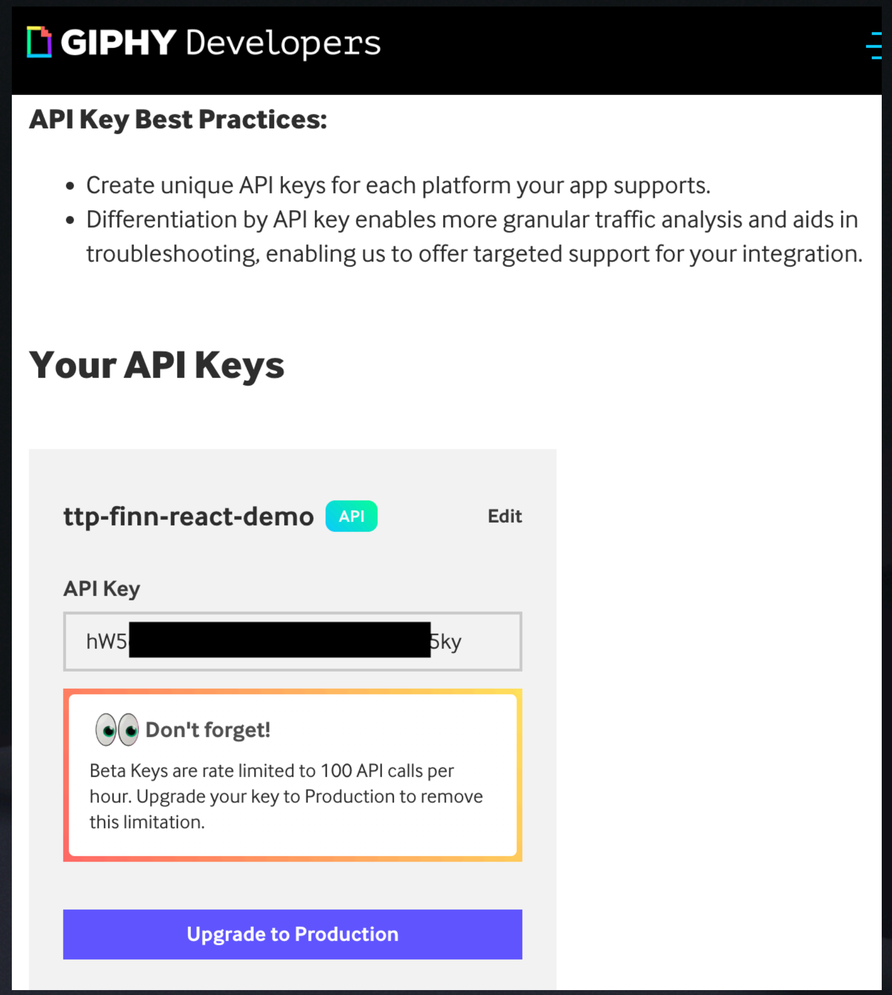
- If anything within the dependencies array changes, React will re-run the useEffect callback function
- If you don’t pass in the second argument to useEffect, it will re-run every time the component renders
  - This is probably NOT what you want

# axios

```
import axios from "axios";

const App = () => {
  const [data, setData] = useState([]);

  const fetchData = async () => {
    const response = await axios.get(url);
    setData(response.data);
  };
  useEffect(() => {
    fetchData();
  }, []);

  return (
    <div>
      {data.map((item) => (
        <div key={item.id}>{item.title}</div>
      ))}
    </div>
  );
};
```

- fetch is fine, but it's annoying to keep having to parse the response data in two different steps
- axios is a popular NPM library that offers more convenient syntax
  - Will come in especially handy when we start making more complicated requests
- Like fetch, it returns a Promise, so remember to await it!

# API Keys




- APIs will often ask you to create an API key, so they know how to apply rate-limits
- Can either be sent in the URL (public) or with an authentication token (private)

```
const url = `http://api.myapp.com/posts?api_key=${YOUR_API_KEY}`;
  const fetchData = async () => {
    const response = await axios.get(url);
    setData(response.data);
  };
  useEffect(() => {
    fetchData();
  }, []);
```

# API Request Practice