Week 6

Last week, our interests lived in a JavaScript file we imported. A merch site can't work like that. It has to ask another service for the current products, and later tell it what goes in the cart. That's what REST APIs are for. By the end, your page will load its cards with a real GET request, handle an empty list and a failure, and you'll know how to read Fourthwall's API docs, the API our merch site will use. We'll request a small JSON file from our own dev server, so nobody needs a shop account or a token.

Getting started

The complete practice files are included at the end of this lesson.

Save and commit your Week 5 work. Open a terminal at the top of `practice-page` and create a branch from that saved work:

```
git status
git switch -c load-interests
```

Copy the supplied Week 6 `App.jsx` and `App.css` into `react-page/src`. Copy `interests.json` into `react-page/public`. Keep the existing `main.jsx`, `styles.css` and last week's `InterestCard.jsx`. The new `App.jsx` still uses `InterestCard` for each card.

This reference includes a finished request function. For the exercise, find `async function loadInterests()` in your copy of `App.jsx`. Replace that entire function with this empty one, leaving the rest of `App` in place:

```
function loadInterests() {
}
```

Start the server from `react-page`:

```
cd react-page
npm run dev
```

Open the address printed by the terminal. The page should show **Load interests** and a message asking you to click. The button won't do anything until we write its function. If `npm` isn't found, check the Setup page.

What an API is

The frontend is what runs in the browser: the page, the buttons, and our React code. A server is a program that gets requests and sends back responses. Our Vite dev server is one. A shop's backend is another.

An API (application programming interface) is the set of requests a server agrees to answer. Its docs tell you what you can send and what you'll get back. You never see the server's code or database, just the requests it accepts.

HTTP is the set of rules browsers and servers use to talk. Every exchange is one **request** from the client and one **response** from the server.

What REST means

REST is a common style for web APIs. It's not something you install. It's a set of habits that make APIs predictable:

1. The API is organized around **resources**, the things it has data about.
2. Each resource has its own **URL**.
3. The **HTTP method** says what you want to do with that resource.
4. The **status code** says how it went.
5. Each request carries everything the server needs to answer it.

Once you know these, most API docs start to look familiar.

Resources and endpoints

A resource is a noun: a product, a collection, a cart. A REST API gives each one an address. For a shop it might look like this:

URL path	Resource
`/products`	All products
`/products/blueprint-shirt`	One product, picked out by its slug
`/carts/abc123`	One cart, picked out by its ID

Paths are nouns, not verbs. You won't usually see `/getProducts`. The verb comes from the method.

An **endpoint** is one method plus one path, like `GET /products`. API docs are mostly a list of endpoints.

Query parameters come after `?`, separated by `&`, like `?page=0&size=50`. They tweak the request, like which page of results you want. They don't change which resource you're asking about.

In our exercise, the resource is the class interest list, and its URL is `/interests.json`.

HTTP methods

The method is the action. These five come up most:

Method	What it asks for	Shop example	Example path
`GET`	Read a resource. Changes nothing.	Get the product list.	`/products`
`POST`	Create something new, or run an action.	Create a new cart.	`/carts`
`PUT`	Replace a whole resource with the one you send.	Replace a saved address.	`/addresses/{addressId}`
`PATCH`	Change only some fields of a resource.	Change just the quantity of one item.	`/carts/{cartId}/items/{itemId}`
`DELETE`	Remove a resource.	Remove one item from a cart.	`/carts/{cartId}/items/{itemId}`

These paths show a typical shop API, not a real one. Check the real docs before using any path.

`GET` is safe to repeat. Asking for the product list twice changes nothing. `POST` usually isn't. Send the same "add to cart" twice and you might get two shirts, so watch out for double clicks.

These are the usual meanings, but the docs have the final say. Some APIs use `POST` where others use `PATCH` or `DELETE`. Fourthwall does this, as we'll see below.

Today we only send `GET`, since our JSON file is just a file and can't store anything. We'll send real `POST` requests when we hook up Fourthwall.

Status codes

Every response starts with a three-digit status code. The first digit tells you the group:

Group	Meaning	Codes you'll see

`2xx`	It worked.	`200 OK`, `201 Created`, `204 No Content`
`4xx`	Something is wrong with the request. Check what you sent.	`400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `409 Conflict`, `429 Too Many Requests`
`5xx`	The server had a problem. Your request might be fine.	`500 Internal Server Error`, `503 Service Unavailable`

Quick rule: 4xx means check your code. 5xx means it's the server, so wait and retry.

Request and response bodies

The body is the data a message carries. A `GET` usually has none, since the URL already says what we want. The response body holds what we asked for.

`POST`, `PUT`, and `PATCH` requests usually have a body saying what to create or change. Most REST APIs use JSON both ways.

Reading JSON

Open `public/interests.json`. The start should look familiar:

```
[
  {
    "id": "games",
    "title": "Video games",
    "description": "I like learning how games are designed."
  }
]
```

JSON (JavaScript Object Notation) is a text format for sending data. It's the same array of objects as last week's `interests.js`, minus the `export`. Names and strings need double quotes, and JSON doesn't allow comments or a trailing comma.

Open `/interests.json` at your app's address, like `http://localhost:5173/interests.json`. Your browser just sent a `GET` for that resource and is showing you the response body. Note `public` isn't part of the URL.

That text still has to become JavaScript data before React can use it. That's what `response.json()` does.

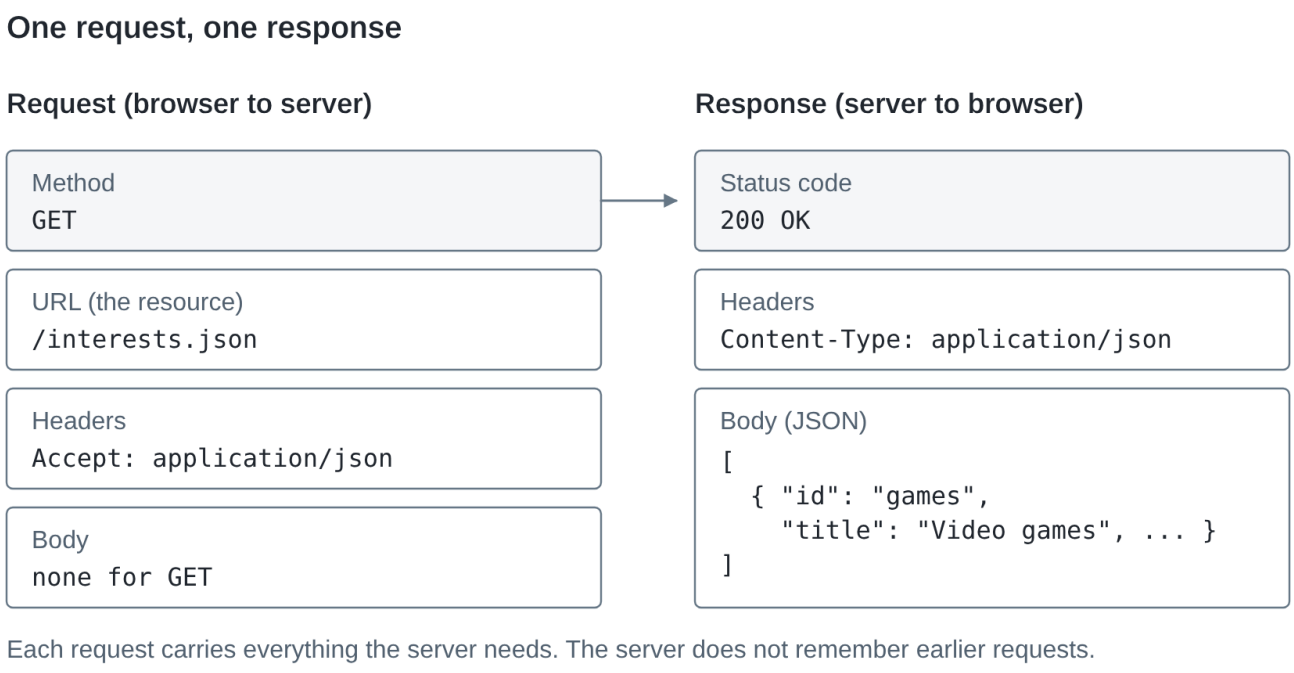
Headers and statelessness

Headers are extra info about a message, separate from the body. A few common ones:

Header	What it says
`Content-Type: application/json`	The body is JSON. Sent on responses, and on requests that have a JSON body.
`Accept: application/json`	The client would like JSON back.
`Authorization`	Proof of who is asking, such as a token. Not every API uses this header. Some put the token in the URL instead.

REST APIs are **stateless**. The server doesn't remember your earlier requests, so each one has to carry everything: method, URL, any token, any body. That's why a cart API wants the cart ID on every cart request. You tell it which cart, every time.

This is about the server. It's not the same as React state, which is the page remembering values between renders.



Each request carries everything the server needs. The server does not remember earlier requests.

Making a GET request from React

Now let's use all that. Our page will send `GET /interests.json`, check the status code, and read the JSON body.

State for a request

Look near the start of `App`. The reference declares three state values:

```
const [interests, setInterests] = useState([]);
const [status, setStatus] = useState("idle");
const [error, setError] = useState("");
```

`interests` starts empty. `status` starts as `"idle"`, meaning we haven't asked yet. `error` holds a message if loading fails.

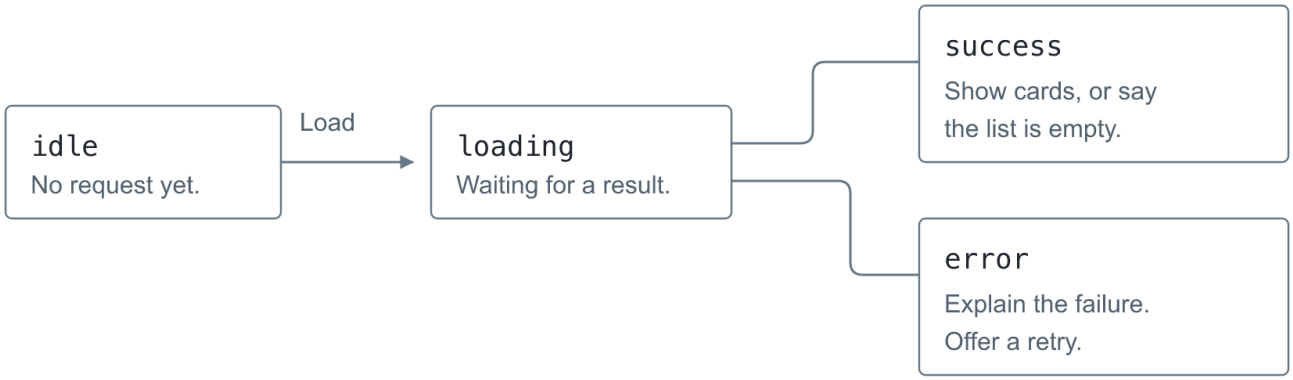
The page uses four status words:

Page status	What someone should see
`idle`	A button and instructions to load the list.
`loading`	A loading message while the request is unfinished.
`success`	Cards, or an explanation that the returned list is empty.
`error`	An error message and a way to try again.

These are our own React state, not the server's HTTP status code.

The page remembers the request result

Our state values describe what the page should show.



An empty array is a successful result. Another Load click starts a new request.

An empty array is still `success`. The request worked, there's just nothing to show. `error` means we never got a usable list.

Waiting for a response

`fetch()` starts a request and gives back a Promise, a value for work that finishes later. An `async` function lets us use `await`, which pauses that function (not the whole browser) until the work is done.

These two lines get and read our response:

```
const response = await fetch("/interests.json");
const data = await response.json();
```

`fetch` sends a `GET` unless you tell it otherwise. The first line gives a Response with the status, headers, and body. The second reads the body as JSON. Getting a response and reading it are two separate steps.

We also have to check the status. `fetch` still hands back a Response for a `404` or `500`. It only fails outright when nothing comes back, like when you're offline. `response.ok` is true for 200 to 299, so `!response.ok` catches the 4xx and 5xx groups.

Handling failures

A few new bits of JavaScript first:

- `if (x) { ... }` runs the braces only when `x` is true.
- `!` means "not", so `!response.ok` means the response failed.
- `throw new Error("...")` makes an error and jumps out.
- `try` wraps code that might fail. If it does, `catch (failure)` runs instead.
- `failure.message` is the error's text. `+` joins strings.
- `return;` just ends the function.

`isInterest` (already in the file) checks one object's fields. `Array.isArray(data)` checks we got a list, and `.every(isInterest)` checks every item. `||` means "or", so if either check fails we reject it.

Replace the empty `loadInterests` function with:

```
async function loadInterests() {
  if (status === "loading") {
    return;
  }

  setStatus("loading");
  setError("");

  try {
    const response = await fetch("/interests.json");
```

```
    if (!response.ok) {
      throw new Error("The request failed with status " + response.status + ".");
    }

    const data = await response.json();

    if (!Array.isArray(data) || !data.every(isInterest)) {
      throw new Error("The response does not contain the interest fields we need.");
    }

    setInterests(data);
    setStatus("success");
  } catch (failure) {
    setError("Could not load interests. " + failure.message);
    setStatus("error");
  }
}
```

If a request is already running, we stop. Otherwise we show loading and clear any old error. A good response with the right fields gets saved and we show success. A failed request, bad JSON, or wrong fields all land in `catch`.

Save and click **Load interests**. The cards should appear.

This is the same update as Week 5's search. The setters ask React to render again, only now the value came from a response instead of a text field. If we never called a setter, nothing on screen would change.

Checking the response's shape

Look at the supplied check near the top of `App.jsx`:

```
function isInterest(value) {
  return (
    value !== null &&
    typeof value === "object" &&
    typeof value.id === "string" &&
    typeof value.title === "string" &&
    typeof value.description === "string"
  );
}
```

It wants an object with three string fields. `typeof` gives a value's type, and `&&` means every check has to pass. (We check for `null` separately because JavaScript says `typeof null` is `"object"`.) This lets us show a clear error instead of crashing. Products will need their own check.

Showing what is happening

The supplied button connects to our function and is disabled while loading:

```
<button type="button" onClick={loadInterests} disabled={status === "loading"}>
  Load interests
</button>
```

Like `onChange` last week, we pass the function without calling it. While loading, `disabled` is true, so nobody can click twice.

The page picks its message with conditions, like week 5's no-match message:

```
{status === "success" && interests.length === 0 && (
  <p>No interests were returned.</p>
)}
```

This shows when loading worked but the list is empty. That's not a failure. In the full file, `role="status"` lets a screen reader announce changes and `role="alert"` announces errors.

Inspecting the request

Open **Inspect**, then **Network**. Click **Load interests** again and find the `interests.json` request.

Match it to the REST terms from earlier:

1. In **Headers**, find the request URL, the method, and the status code. The URL is the resource. The method should be `GET` and the status should be `200`.
2. Still in **Headers**, scroll to the response headers and find `Content-Type`. It should say the body is JSON.
3. In **Response** or **Preview**, find the response body: the returned array and its objects. This is the data our function passed to `setInterests`.
4. Compare a returned title with its card on the page.

If it loads too fast to see, turn on throttling in the Network panel and click again. You should see the loading message and a disabled button. Turn throttling off after. Nothing loads until you click, since the click handler starts the request.

Try an empty list and a failure

Save a copy of the working JSON. Then replace it with:

```
[]
```

Save and click the button. You should get **No interests were returned**. Restore the data.

Next, change `"title"` to `"heading"` in one object. It's still valid JSON, just the wrong shape. Load it, read the error, fix the name, and retry without refreshing.

Then pick **Offline** in the Network panel and click. You should get the error. Switch back to normal after.

Last, change the URL in `loadInterests` to `"/missing.json"` and click. Check the row in Network. Vite sometimes answers missing paths with its own HTML page and a `200`, so you might get a JSON error instead of a `404`. Either way the page shows an error, which is why we check both the status and the shape. Put the URL back.

Reading API docs: Fourthwall

Our merch site will use Fourthwall's Storefront API. Its docs follow the same REST habits, so let's read them like any new API. No token needed, we're just reading.

Here are the endpoints the project will use:

Endpoint	What it does
`GET /v1/collections/all/products`	Read the products in the `all` collection
`GET /v1/products/{slug}`	Read one product by its slug
`POST /v1/carts`	Create a new cart
`GET /v1/carts/{cartId}`	Read one cart
`POST /v1/carts/{cartId}/add`	Add items to a cart
`POST /v1/carts/{cartId}/change`	Change item quantities
`POST /v1/carts/{cartId}/remove`	Remove items

These come from Fourthwall's Storefront API schema, checked November 2, 2026. Each one has its own reference page, for example Get Product, Create Cart, Get Cart, and Add to Cart.

`{slug}` and `{cartId}` are placeholders for real values. Every path starts from `https://storefront-api.fourthwall.com`.

Look at the methods. Reading is `GET`, but every cart change is `POST` with an action word at the end. Another API might use `PATCH` or `DELETE` here. That's why you read the docs instead of guessing.

A full product-list request looks like this:

```
GET https://storefront-api.fourthwall.com/v1/collections/all/products?storefront_token=YOUR_TOKEN&currency=USD&page=0&size=50
```

This isn't a command to run, just a request to read. `YOUR_TOKEN` stands for the shop's Storefront token, and since the API is stateless, it goes on every request. Fourthwall puts it in a `storefront_token` query parameter rather than an `Authorization` header (Storefront getting started). `currency` picks the price currency, and `page` and `size` pick which slice of products comes back.

The response body, described in Fetching Products, is an object containing `results` and `paging`, not a bare array. `results` holds the products. `paging` describes the page you got: `pageNumber`, `pageSize`, `elementsSize`, `elementsTotal`, `totalPages`, and `hasNextPage`. This shortened example uses made-up values and skips most product fields:

```
{
  "results": [
    {
      "type": "PRODUCT",
      "id": "example-product",
      "name": "Blueprint shirt",
      "slug": "blueprint-shirt",
      "variants": []
    }
  ],
  "paging": {
    "hasNextPage": false
  }
}
```

So the products are at `data.results`, and the first name is `data.results[0].name`. Our interests use `title`, Fourthwall uses `name`. Each variant has an ID and a `unitPrice`, which we'll need for the cart.

The docs list errors too. A slug that doesn't exist on `GET /v1/products/{slug}` returns `404` with a body whose `code` is `OFFER_SLUG_NOT_FOUND_ERROR`. Good APIs tell you what went wrong, not just the number.

So just swapping the URL in our exercise wouldn't work. We'd need a check for product fields, handle more pages, and watch `type`, since bundles come back too.

Open Fourthwall's Get Product page (under Resources) and write down these five things. It works for any API:

1. The method and path.
2. The required parameters, and where each goes (path, query, or body).
3. The fields in a successful response.
4. The error status codes it can return.
5. How it expects you to prove access, such as a token.

One more thing: Fourthwall is a different site from ours, so the browser only lets us read its responses because Fourthwall allows it (this rule is called CORS). Not every API does, and `mode: "no-cors"` won't fix it.

When something goes wrong

What you see	What to check
No request in Network	Confirm the button uses `onClick={loadInterests}` and the function contains `fetch`.
JSON parsing error or an HTML response	Open the exact request URL. Confirm the file is `public/interests.json`, not `src/interests.json` or `public/public/interests.json`.
Wrong-field error	Check `id`, `title`, and `description` and confirm that the outer value is an array.
The page stays in loading	Check the Console, keep both success and catch status updates, and make sure you restored normal network settings.
HTTP `401` or `403` on a future API call	Read the response and check the service's access instructions and configured token.
HTTP `404`	Check the endpoint and resource ID. A missing local path may instead return Vite's HTML page, which causes a JSON parsing error.
HTTP `429`	Too many requests were made. Follow the API's retry guidance instead of repeatedly clicking.
HTTP `500`	The server reported an error. Keep the error message and retry path useful; changing the page's CSS will not fix the server.

The meanings of other HTTP codes are listed on the MDN status codes page under Resources.

What's next

Before next week, restore the JSON and the URL, and turn off throttling and Offline. Add one interest to the JSON, load it, and check its card shows up. Commit with a message such as `Load interests with a GET request` and push the branch. If GitHub asks you to sign in, use the GitHub CLI steps on the Setup page.

Next week, we'll use state to pick a product variant and update a small practice cart. Later, those cart changes become the `POST` requests above.

Resources

Everything needed for today's code and explanations is above. These are extra examples or references to return to after the exercise.

- MDN: Overview of HTTP
- MDN: Working with JSON
- MDN: Using Fetch
- MDN: async functions
- MDN: try...catch
- MDN: Response.ok
- MDN: typeof
- MDN: CORS
- MDN: HTTP request methods
- MDN: HTTP headers
- MDN: HTTP status codes
- Fourthwall: Fetching Products
- Fourthwall: Get Product
- Fourthwall: Storefront getting started
- Fourthwall: Storefront API schema

Complete practice files

Copy these files into the paths shown below. They are the same files used by the lesson. Keep the filenames and imports together. Keep last week's `InterestCard.jsx` (or copy the version below if you need it).

src/InterestCard.jsx

```
export default function InterestCard(props) {
  return (
    <article className="interest-card">
      <h3>{props.title}</h3>
      <p>{props.description}</p>
    </article>
  );
}
```

src/App.jsx

```
import { useState } from "react";
import InterestCard from "./InterestCard.jsx";
import "./App.css";

function isInterest(value) {
  return (
    value !== null &&
    typeof value === "object" &&
    typeof value.id === "string" &&
    typeof value.title === "string" &&
    typeof value.description === "string"
  );
}

function renderInterest(interest) {
  return (
    <InterestCard
      key={interest.id}
      title={interest.title}
      description={interest.description}
    />
  );
```

```
}

export default function App() {
  const [interests, setInterests] = useState([]);
  const [status, setStatus] = useState("idle");
  const [error, setError] = useState("");

  async function loadInterests() {
    if (status === "loading") {
      return;
    }

    setStatus("loading");
    setError("");

    try {
      const response = await fetch("/interests.json");

      if (!response.ok) {
        throw new Error("The request failed with status " + response.status + ".");
      }

      const data = await response.json();

      if (!Array.isArray(data) || !data.every(isInterest)) {
        throw new Error("The response does not contain the interest fields we need.");
      }

      setInterests(data);
      setStatus("success");
    } catch (failure) {
      setError("Could not load interests. " + failure.message);
      setStatus("error");
    }
  }

  return (
    <main>
      <h1>Load my interests</h1>
      <p>
```

```
        This practice page requests a JSON file from the same server as the page.
      </p>
      <button type="button" onClick={loadInterests} disabled={status === "loading"}>
        Load interests
      </button>

      <div role="status">
        {status === "idle" && <p>Click the button to request the list.</p>}
        {status === "loading" && <p>Loading interestsâ€¦</p>}
        {status === "success" && interests.length > 0 && (
          <p>Interests loaded: {interests.length}</p>
        )}
        {status === "success" && interests.length === 0 && (
          <p>No interests were returned.</p>
        )}
      </div>

      {status === "error" && (
        <p role="alert">{error} Check the Network tab, then click Load interests to try
again.</p>
      )}
      {status === "success" && (
        <div className="interest-list">{interests.map(renderInterest)}</div>
      )}
    </main>
  );
}
```

src/App.css

```
.interest-card h3 {
  font-size: 1.15rem;
  margin: 0;
}

button:disabled {
  background: #475569;
  cursor: wait;
}
```

```
[role="alert"] {
  color: #991b1b;
  border-left: 4px solid #b91c1c;
  padding: 12px;
  background: #fef2f2;
}
```

public/interests.json

```
[
  {
    "id": "games",
    "title": "Video games",
    "description": "I like learning how games are designed."
  },
  {
    "id": "photos",
    "title": "Photography",
    "description": "I like taking pictures around Hoboken."
  },
  {
    "id": "cooking",
    "title": "Cooking",
    "description": "I like trying a new recipe each week."
  }
]
```

Revision #7
Created 2026-11-02 18:56:36 UTC by Nishit Sharma
Updated 2026-11-04 18:36:35 UTC by Nishit Sharma