🏏 Requirements

# **Neurolabs Frontend Engineer Technical Test - 2025 [Shared]**

## **Welcome**

Hi there,

Thanks for your interest in the Neurolabs Frontend Engineer role!

This is a simple technical test to give us an opportunity to see your technical skills in action and give you a chance to use our API.

<aside> ‼️ Please don’t spend more than **a few hours** working on this.

</aside>

We will review the test and should you pass this stage, we will use it a basis for a discussion around your technical skills and some of the challenges we face.

## **Background**

We have developed an Image Recognition API that our clients integrate with to recognise products in images. There are two key concepts: a Catalog of *items* and Image Recognition (IR) *tasks*.

For this test, we have:

* Created a test user account with a set of example Catalog items.
* Created an IR task for detecting those items.

View our Staging API docs [here](https://staging.api.neurolabs.ai/v2/docs#/).

Using the API docs, you can authorise with your API key and test endpoints.

<aside> ‼️ We will give you an API key for a test user in our Staging environment. This will be revoked once you return your test. **Please use it sensibly!**

</aside>

## **Task 1: Visualise the Catalog**

1. To view the available Catalog items:
   1. GET /v2/catalog-items
2. Create an App to integrate with the API
   1. Using React and Typescript
3. Query for the items in the Catalog
4. Visualise the Catalog items
   1. Including the thumbnail
5. Highlight the Catalog item status
   1. In particular, those that need ‘capture’

## **Task 2: Submit IR tasks**

1. To view the available IR tasks:
   1. GET /v2/image-recognition/tasks
   2. Find a task and copy the uuid (task\_uuid)
2. Add functionality to the App to submit images to the IR tasks
3. Open an image to send
   1. Show the image
4. Submit the image to the endpoint
   1. POST **/v2/image-recognition/tasks/{task\_uuid}/images**
   2. Show the state of the request
5. Add ability to send further requests to view the status of the original request
   1. Show the output/result
6. Include a few tests

<aside> ‼️ If there are CORS issues, one approach is to add "proxy": "<https://staging.api.neurolabs.ai/>" to the package.json and remove this base URL from any URLs.

</aside>

## **Finally**

Any feedback about the docs, API or this test is most welcome!

Best of luck!

Many thanks,

Neurolabs tech team

## **API integration**

GET /v2/catalog-items  
  
 {

"items": [

{

"uuid": "3fa85f64-5717-4562-b3fc-2c963f66afa6",

"status": "READY",

"thumbnail\_url": "https://example.com/",

"name": "string",

"barcode": "string",

"custom\_id": "string",

"height": 0,

"width": 0,

"depth": 0,

"brand": "string",

"size": "string",

"container\_type": "string",

"flavour": "string",

"packaging\_size": "string",

"created\_at": "2025-05-31T13:08:15.418Z",

"updated\_at": "2025-05-31T13:08:15.418Z"

}

],

"total": 0,

"limit": 1,

"offset": 0

}

GET /v2/image-recognition/tasks

{

"items": [

{

"uuid": "3fa85f64-5717-4562-b3fc-2c963f66afa6",

"name": "string",

"created\_at": "2025-05-31T13:12:17.481Z",

"updated\_at": "2025-05-31T13:12:17.481Z",

"compute\_realogram": true,

"compute\_shares": true

}

],

"total": 0,

"limit": 1,

"offset": 0

}

POST /v2/image-recognition/tasks/{task\_uuid}/images

This endpoint will start image recognition using existing image recognition task.

Images need to be uploaded using the multipart/form-data encoding method.

A separate result will be created for each specified image and an array of their uuids will be returned as the response.

This endpoint works asynchronously: it will return a response right away, but image recognition will take some time.

You can track their statuses via /v2/image-recognition/tasks/{task\_uuid}/results/{result\_uuid} endpoint.

If you provide a value for the callback field in the original payload, after recognition is ready, an HTTP POST request will be sent to the specified url with the result as a payload.

A callback request will be sent for each result independently as soon as it is ready.

Parameters

Callbacks

Try it out

Name Description

task\_uuid \*

string($uuid)

(path)

task\_uuid

callback

string | (string | null)($uri)

(query)

callback

minLength: 1

maxLength: 2083

Request body

multipart/form-data

images \*

array<string>

Responses

Code Description Links

200

Successful Response

Media type

application/json

Controls Accept header.

Example Value

Schema

[

"3fa85f64-5717-4562-b3fc-2c963f66afa6"

]

No links

422

Validation Error

Media type

application/json

Example Value

Schema

{

"detail": [

{

"loc": [

"string",

0

],

"msg": "string",

"type": "string"

}

]

}

🏗️ Requirements Architecture

# **Neurolabs Frontend Technical Test Architecture**

## **Overview**

This document outlines the architecture for the Neurolabs Frontend Engineer Technical Test application. The application integrates with the Neurolabs Staging API to visualize catalog items and submit image recognition (IR) tasks. The solution is built using React with TypeScript, styled with Tailwind CSS, and includes unit tests with Jest and React Testing Library. The application features a landing page with navigation to a Home screen, where all test-related functionality is centralized.

## **Objectives**

* **Visualize Catalog Items**: Display catalog items with thumbnails and highlight items needing 'capture' status.
* **Submit IR Tasks**: Allow users to view IR tasks, upload images, submit them to a task, and check the status of submissions.
* **Ensure Code Quality**: Use TypeScript for type safety, modular components, and include tests for critical functionality.
* **Handle CORS**: Configure a proxy in package.json to resolve CORS issues with the Staging API.
* **Provide a User-Friendly Entry Point**: Include a landing page with navigation to the main functionality.

## **Tech Stack**

* **Frontend Framework**: React 18 (with JSX)
* **Language**: TypeScript
* **Styling**: Tailwind CSS (via CDN for simplicity)
* **HTTP Client**: Native Fetch API (for API requests)
* **Routing**: React Router DOM
* **Testing**: Jest, React Testing Library
* **Build Tool**: Create React App (for development and bundling)
* **API**: Neurolabs Staging API (https://staging.api.neurolabs.ai/)

## **System Architecture**

The application is a single-page application (SPA) with a modular component-based architecture, using React Router for navigation. It follows a clean architecture pattern with separation of concerns between UI components, API services, and state management. The entry point is a landing page (/) that navigates to the Home screen (/home), where all test functionality is centralized.

### **Components**

* **App**: Root component that sets up React Router with two routes:
  + /: Landing page with a styled welcome message and a button to navigate to /home.
  + /home: Renders the Home component, which contains all test-related components.
* **Home**: Central component that integrates CatalogView, IRTaskView, and supporting components to fulfill test requirements.
* **CatalogView**: Displays the list of catalog items with thumbnails and status indicators.
* **IRTaskView**: Manages IR task selection, image upload, submission, and status display.
* **CatalogItemCard**: Reusable component to render individual catalog item details.
* **ImageUploader**: Handles image selection and preview.
* **TaskStatusDisplay**: Shows the status and results of IR task submissions.
* **LoadingSpinner**: Reusable component for loading states.
* **ErrorMessage**: Reusable component for error states.

### **Services**

* **ApiService**: Encapsulates API calls to the Neurolabs Staging API using the Fetch API.
  + getCatalogItems: Fetches catalog items (GET /v2/catalog-items).
  + getIRTasks: Fetches available IR tasks (GET /v2/image-recognition/tasks).
  + submitImage: Submits an image to an IR task (POST /v2/image-recognition/tasks/{task\_uuid}/images).
  + getTaskStatus: Polls the status of an IR task submission.
* **Config**: Stores API key and base URL, loaded from environment variables.

### **State Management**

* **React Context**: Used for global state (e.g., API key, selected task UUID).
* **React Hooks**:
  + useState: Manages local component state (e.g., selected image, loading state).
  + useEffect: Handles side effects like API calls and polling.
  + useQuery (from react-query, optional): Manages API data fetching and caching (if added for optimization).

### **Data Flow**

1. **Landing Page**:
   * The App component renders a landing page at / with a button to navigate to /home.
   * Navigation uses react-router-dom to load the Home component.
2. **Catalog Visualization** (in Home):
   * CatalogView triggers ApiService.getCatalogItems on mount.
   * Response data is stored in the component state.
   * CatalogItemCard components render each item, with conditional styling for 'capture' status.
3. **IR Task Submission** (in Home):
   * IRTaskView fetches tasks via ApiService.getIRTasks.
   * User selects a task UUID and uploads an image via ImageUploader.
   * ApiService.submitImage sends the image to the API.
   * TaskStatusDisplay polls ApiService.getTaskStatus to show results.
4. **Error Handling**:
   * API errors are caught and displayed using ErrorMessage.
   * Network failures trigger retry logic (limited to 3 attempts).

### **File Structure**

neurolabs-test/

├── src/

│ ├── components/

│ │ ├── CatalogItemCard.tsx

│ │ ├── CatalogView.tsx

│ │ ├── Home.tsx

│ │ ├── ImageUploader.tsx

│ │ ├── IRTaskView.tsx

│ │ ├── LoadingSpinner.tsx

│ │ ├── ErrorMessage.tsx

│ │ └── TaskStatusDisplay.tsx

│ ├── services/

│ │ ├── ApiService.ts

│ │ └── Config.ts

│ ├── context/

│ │ └── AppContext.tsx

│ ├── types/

│ │ └── index.ts

│ ├── App.tsx

│ ├── index.tsx

│ ├── index.css

│ └── reportWebVitals.ts

├── public/

│ ├── index.html

│ ├── favicon.ico

│ ├── logo192.png

│ ├── logo512.png

│ ├── manifest.json

│ └── robots.txt

├── package.json

├── tsconfig.json

└── README.md

## **API Integration**

* **Base URL**: https://staging.api.neurolabs.ai/
* **Authentication**: API key passed via Authorization: Bearer header.
* **CORS Handling**: Proxy set in package.json ("proxy": "https://staging.api.neurolabs.ai/").
* **HTTP Client**: Fetch API is used for all requests, with FormData for image uploads and JSON parsing for responses.
* **Endpoints and 200 OK Response Structures**:
  1. **GET /v2/catalog-items**:
     + **Purpose**: Fetches the list of catalog items.

**Response (200 OK)**:  
[

{

"id": "string", // Unique identifier for the catalog item

"name": "string", // Name of the item

"thumbnail": "string", // URL to the item's thumbnail image

"status": "string", // Status of the item (e.g., "capture", "active", "inactive")

"description": "string", // Optional detailed description of the item

"category": "string", // Optional category or type of the item (e.g., "beverage", "snack")

"created\_at": "string", // ISO 8601 timestamp of item creation (e.g., "2025-05-31T15:21:00Z")

"updated\_at": "string", // ISO 8601 timestamp of last update (e.g., "2025-05-31T15:21:00Z")

"metadata": { // Optional key-value pairs for additional item details

"sku": "string", // Stock keeping unit or product code

"brand": "string", // Brand name

"weight": number, // Weight in grams or other unit

"dimensions": { // Physical dimensions

"width": number, // Width in cm

"height": number, // Height in cm

"depth": number // Depth in cm

}

},

"image\_count": number // Number of images associated with the item for recognition

},

...

]

* + - **Notes**: The response is an array of catalog items. Required fields are id, name, thumbnail, and status, used for visualization and status highlighting (e.g., 'capture'). Optional fields (description, category, created\_at, updated\_at, metadata, image\_count) provide additional context for display or filtering. The thumbnail URL is assumed to be publicly accessible. The status field drives conditional styling in CatalogItemCard.
  1. **GET /v2/image-recognition/tasks**:
     + **Purpose**: Fetches the list of available IR tasks.

**Response (200 OK)**:  
[

{

"uuid": "string", // Unique identifier for the IR task

"name": "string" // Name or description of the task

},

...

]

* + - **Notes**: The response is an array of IR tasks. The uuid is used for submitting images to a specific task.
  1. **POST /v2/image-recognition/tasks/{task\_uuid}/images**:
     + **Purpose**: Submits an image to the specified IR task.
     + **Request Body**: FormData containing the image file (key: image).

**Response (200 OK)**:  
{

"image\_id": "string", // Unique identifier for the submitted image

"task\_uuid": "string", // UUID of the task the image was submitted to

"status": "string" // Initial status of the submission (e.g., "pending")

}

* + - **Notes**: The image\_id is used to poll for the submission status. The status field indicates the processing state.
  1. **GET /v2/image-recognition/tasks/{task\_uuid}/images/{image\_id}** (Assumed):
     + **Purpose**: Retrieves the status and results of an image submission.

**Response (200 OK)**:  
{

"image\_id": "string", // Unique identifier for the submitted image

"task\_uuid": "string", // UUID of the task

"status": "string", // Current status (e.g., "pending", "completed", "failed")

"result": {

// Optional result data if status is "completed"

"recognized\_items": [

{

"item\_id": "string", // ID of recognized catalog item

"confidence": number // Confidence score (0 to 1)

},

...

]

}

}

* + - **Notes**: This endpoint is assumed based on standard REST patterns for polling task status. The result field is included only when the status is "completed".

## **Styling**

* **Tailwind CSS**: Used for responsive, utility-first styling.
* **Layout**: Flexbox and Grid for catalog and task views within the Home component.
* **Theming**: Consistent colors and typography (e.g., blue for active states, red for errors).
* **Landing Page**: Gradient background with animated fade-in text and a styled button, as defined in App.tsx.

## **Testing**

* **Unit Tests**:
  + CatalogView: Renders catalog items correctly, including optional fields like description and category.
  + ImageUploader: Handles file selection and preview.
  + ApiService: Mocks Fetch API calls to ensure correct request/response handling.
  + Home: Verifies rendering of child components (CatalogView, IRTaskView).
  + App: Tests routing to landing page and Home screen.
* **Tools**: Jest, React Testing Library (included with Create React App).
* **Coverage**: Focus on critical components and services.

## **Deployment**

* **Development**: Run locally with npm start (Create React App).
* **Build**: npm run build generates a production-ready bundle.
* **Dependencies**: Managed via npm, listed in package.json.

## **Assumptions**

* API key is provided and stored in .env as REACT\_APP\_API\_KEY.
* IR task status endpoint is assumed based on standard REST patterns.
* Thumbnail URLs in catalog items are accessible without additional authentication.
* Image uploads are handled as multipart/form-data via Fetch API.
* The Home component is responsible for orchestrating all test-related functionality.
* The GET /v2/catalog-items response structure is inferred from the test description and typical REST API conventions for catalog management in image recognition systems, as the exact schema was not provided.
* Optional fields in the GET /v2/catalog-items response (description, category, created\_at, updated\_at, metadata, image\_count) may be null or omitted if not applicable.

## **Risks and Mitigations**

* **CORS Issues**: Mitigated by proxy configuration.
* **API Rate Limits**: Handle with retry logic and exponential backoff.
* **Type Safety**: Use TypeScript interfaces for API responses, aligned with documented response structures, with optional fields to handle variability.
* **Scalability**: Modular components and services allow easy extension.
* **Routing Performance**: React Router is lightweight, but lazy loading could be added for larger apps.
* **API Schema Changes**: TypeScript interfaces can be updated if actual response structures differ from assumed properties.

## **Future Improvements**

* Add react-query for advanced data fetching and caching.
* Implement pagination for large catalog lists.
* Add accessibility (ARIA) attributes for better usability.
* Enhance tests with end-to-end testing (e.g., Cypress).
* Introduce lazy loading for routes to optimize performance.
* Display additional catalog item properties (e.g., description, category) in CatalogItemCard for richer visualization.

🎖️ Code based Architecture

# **Neurolabs Frontend Engineer Technical Test Architecture**

## **Architectural Summary**

* **Framework and Language**: Built with React (using JSX) and TypeScript for type-safe, component-based development.
* **State Management**: Utilizes React Context (AppContext) for global API key management, with local component state via useState for UI interactions.
* **Routing**: Implements client-side routing with react-router-dom for navigation between landing page and main demo page.
* **API Integration**: Communicates with Neurolabs Staging API via a proxy (http-proxy-middleware) to handle CORS, using fetch for HTTP requests.
* **UI Components**: Modular, reusable components (CatalogView, IRTaskView, ImageUploader, etc.) styled with Tailwind CSS for responsive design.
* **Error Handling**: Centralized error management in ApiService with user-friendly error messages displayed via ErrorMessage component.
* **Loading States**: Visual feedback during API calls using LoadingSpinner component.
* **Testing**: Comprehensive unit tests using Jest and React Testing Library, covering API services, components, and context.
* **Logging**: Custom logging in ApiService and proxy setup for debugging, with environment-based log control.
* **Type Safety**: Strongly typed interfaces (CatalogItem, IRTask, etc.) defined in types/index.ts for consistent data handling.
* **Proxy Setup**: Custom proxy middleware (setupProxy.js) logs requests and handles API routing in development.
* **Environment Configuration**: Uses .env for API key and base URL, with runtime validation in Config.ts.

## **Component Structure**

| **Component** | **Purpose** | **Key Features** |
| --- | --- | --- |
| App | Root component with routing setup | Routes to landing page and Home |
| Home | Main demo page hosting catalog and IR task views | Navigation back to landing, grid layout |
| CatalogView | Displays catalog items fetched from API | Grid display, loading/error states |
| CatalogItemCard | Renders individual catalog item with thumbnail and status | Conditional styling based on status |
| IRTaskView | Manages IR task selection, image upload, and status display | Task dropdown, integrates ImageUploader |
| ImageUploader | Handles image selection, preview, and upload submission | File input, preview, upload/reset buttons |
| TaskStatusDisplay | Polls and displays status of uploaded image | Auto-polling, retry/refresh buttons |
| LoadingSpinner | Visual indicator for loading states | Animated spinner with Tailwind styling |
| ErrorMessage | Displays error messages | Centered, red-styled error text |

## **File Structure**

| **File/Folder** | **Description** |
| --- | --- |
| src/index.tsx | Entry point, renders App with AppProvider |
| src/App.tsx | Defines routes for landing and demo pages |
| src/components/ | Contains all UI components (e.g., CatalogView, ImageUploader) |
| src/services/ | API service (ApiService.ts) and config (Config.ts) |
| src/context/ | AppContext.tsx for API key management |
| src/types/ | TypeScript interfaces (CatalogItem, IRTask, etc.) |
| src/tests/ | Jest unit tests for components and services |
| src/setupProxy.js | Proxy middleware for API requests in development |
| src/App.css, index.css | Global styles (minimal, as Tailwind is primary) |
| src/setupTests.ts | Jest DOM setup for testing |

## **API Integration**

| **Endpoint** | **Method** | **Purpose** | **Component/Service** |
| --- | --- | --- | --- |
| /v2/catalog-items | GET | Fetch catalog items | CatalogView/ApiService |
| /v2/image-recognition/tasks | GET | Fetch IR tasks | IRTaskView/ApiService |
| /v2/image-recognition/tasks/{task\_uuid}/images | POST | Submit image for IR task | ImageUploader/ApiService |
| /v2/image-recognition/tasks/{task\_uuid}/images/{image\_id} | GET | Check image processing status | TaskStatusDisplay/ApiService |

## **Testing Coverage**

| **Test File** | **Coverage Area** | **Key Scenarios Tested** |
| --- | --- | --- |
| ApiService.test.ts | API service functions | Fetch catalog items, IR tasks, image upload, status |
| AppContext.test.ts | Context provider and hook | API key provision, usage outside provider |
| ImageUploader.test.tsx | Image upload component | File selection, preview, upload, reset, styling |
| IRTaskView.test.tsx | IR task view component | Loading, error, task rendering, reset |
| TaskStatusDisplay.test.tsx | Task status component | Loading, success, error, polling, styling |
| LoadingSpinner.test.tsx | Loading spinner component | DOM hierarchy, styling |
| ErrorMessage.test.tsx | Error message component | Message rendering, styling, empty message |
| CatalogView.test.tsx | Catalog view component | Loading, error, item rendering |
| CatalogItemCard.test.tsx | Catalog item card component | Item details, status styling, optional fields |

## **Styling Approach**

* **Primary Styling**: Tailwind CSS for utility-first, responsive design.
* **Status Highlighting**: Dynamic classes in CatalogItemCard (e.g., border-green-500 for READY, border-red-500 for PROCESSING).
* **Animations**: Tailwind transitions for hover effects (hover:scale-105, hover:shadow-lg) and loading spinner (animate-spin).
* **Minimal Custom CSS**: App.css and index.css used sparingly for global overrides.

## **Key Features**

* **Catalog Visualization**: Displays items with thumbnails, highlights capture status with red border and background.
* **IR Task Submission**: Allows task selection, image upload with preview, and status polling with retry/refresh options.
* **Responsive Design**: Grid layouts adjust for mobile and desktop using Tailwind’s responsive classes.
* **Type Safety**: Ensures robust data handling with TypeScript interfaces.
* **Testing**: Extensive unit tests ensure component and API reliability.
* **CORS Handling**: Proxy setup mitigates CORS issues in development.

🥌 Presentation

<https://github.com/atafs/technical-test-react-typescript-api-jest/tree/main/architecture/presentation>

🎁 Releases

# Repos

FE: <https://github.com/atafs/technical-test-react-typescript-api-jest>  
  
BE mock staging API: <https://github.com/atafs/technical-test-server-mock-contract>

# Demos

Demo task 1:   
<https://drive.google.com/file/d/1m18PuRHZT-bQ58MnL90_56eOTO_uZ7-R/view?usp=drive_link>

Demo task 2: <https://drive.google.com/file/d/1H7_m81Z3KtM7dXIpImv4FWWxkBMr93ui/view?usp=drive_link>

# Releases and Notes

### Front End

* <https://github.com/atafs/technical-test-react-typescript-api-jest/releases/tag/task1>
* <https://github.com/atafs/technical-test-react-typescript-api-jest/releases/tag/task2>

### Back End

* <https://github.com/atafs/technical-test-server-mock-contract/releases/tag/task1>
* <https://github.com/atafs/technical-test-server-mock-contract/releases/tag/task2>

Final Release pointing a staging url (FE only)

<https://github.com/atafs/technical-test-react-typescript-api-jest/releases/tag/task_final>