



Testing Slim Framework Actions

All in all, it's not too complicated to test a Slim action, and `Environment::mock()` makes it easy to set up particular test cases. Let me show you how it's done.



by Rob Allen MVB · Mar. 20, 16 · Web Dev

Zone · Tutorial



Like (5)



Comment (0)



Save



Tweet



2,892 Views

Building real-time chat? [Enroll in a Free Course on Mobile Chat Development.](#)

To test a [Slim Framework](#) action, you need a request and a response object, and mock whatever is in the action. This is one way to do this.

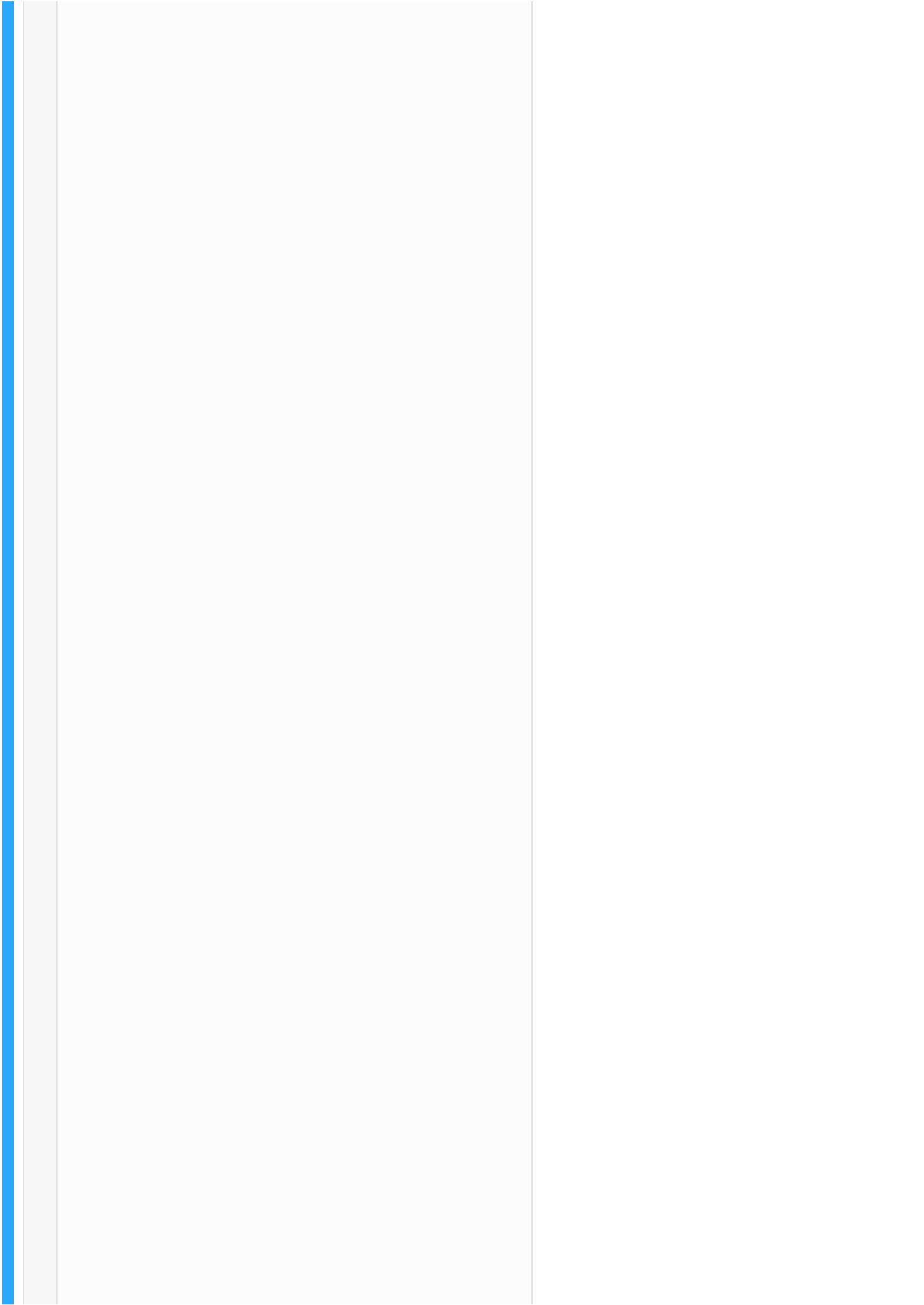
Consider this simple echo action that returns the query parameters to you as a JSON encoded string:

```
1 $ curl "http://localhost:8888/echo?foo=bar&this=that"
2 {"foo":"bar","this":"that"}
```



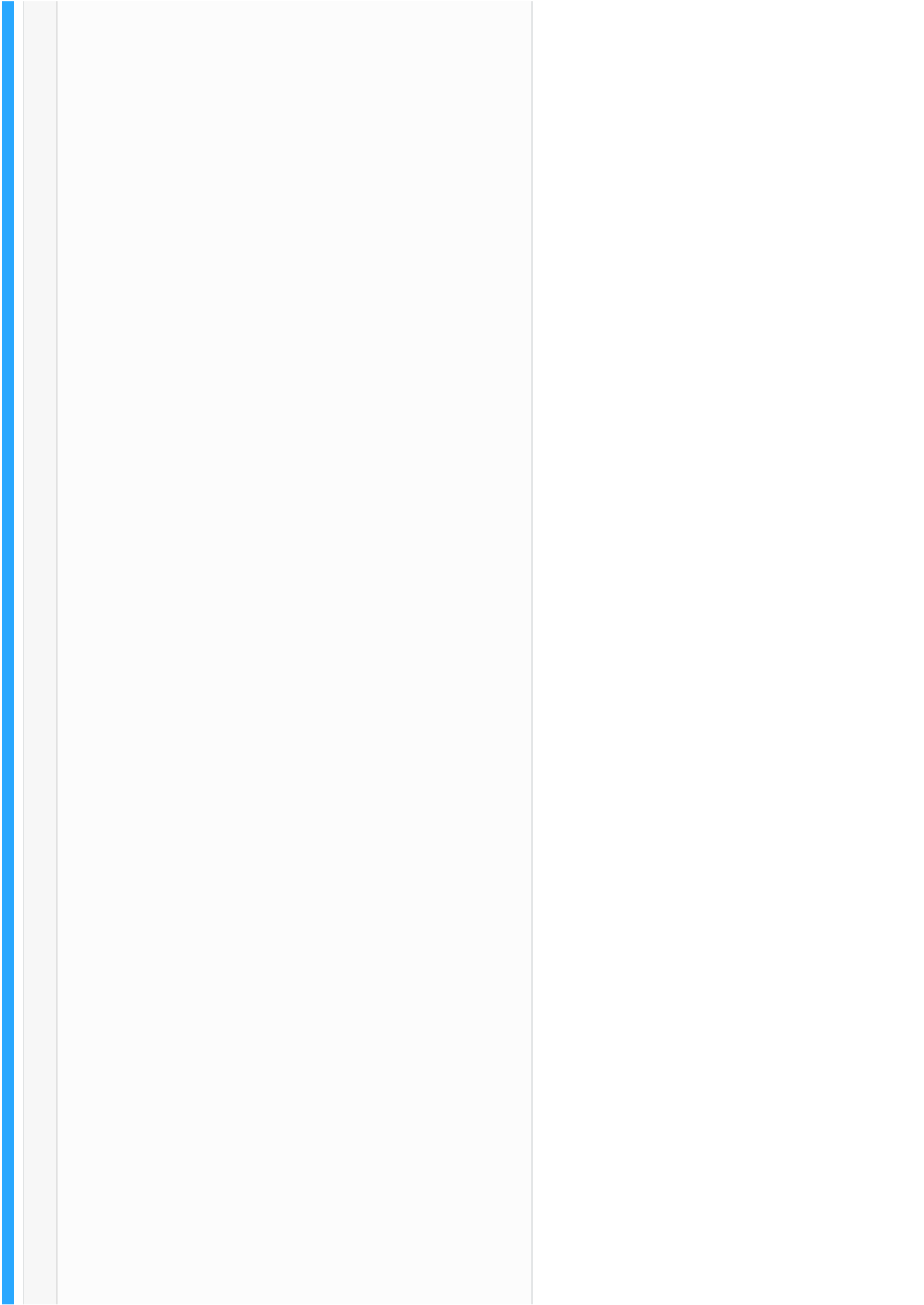


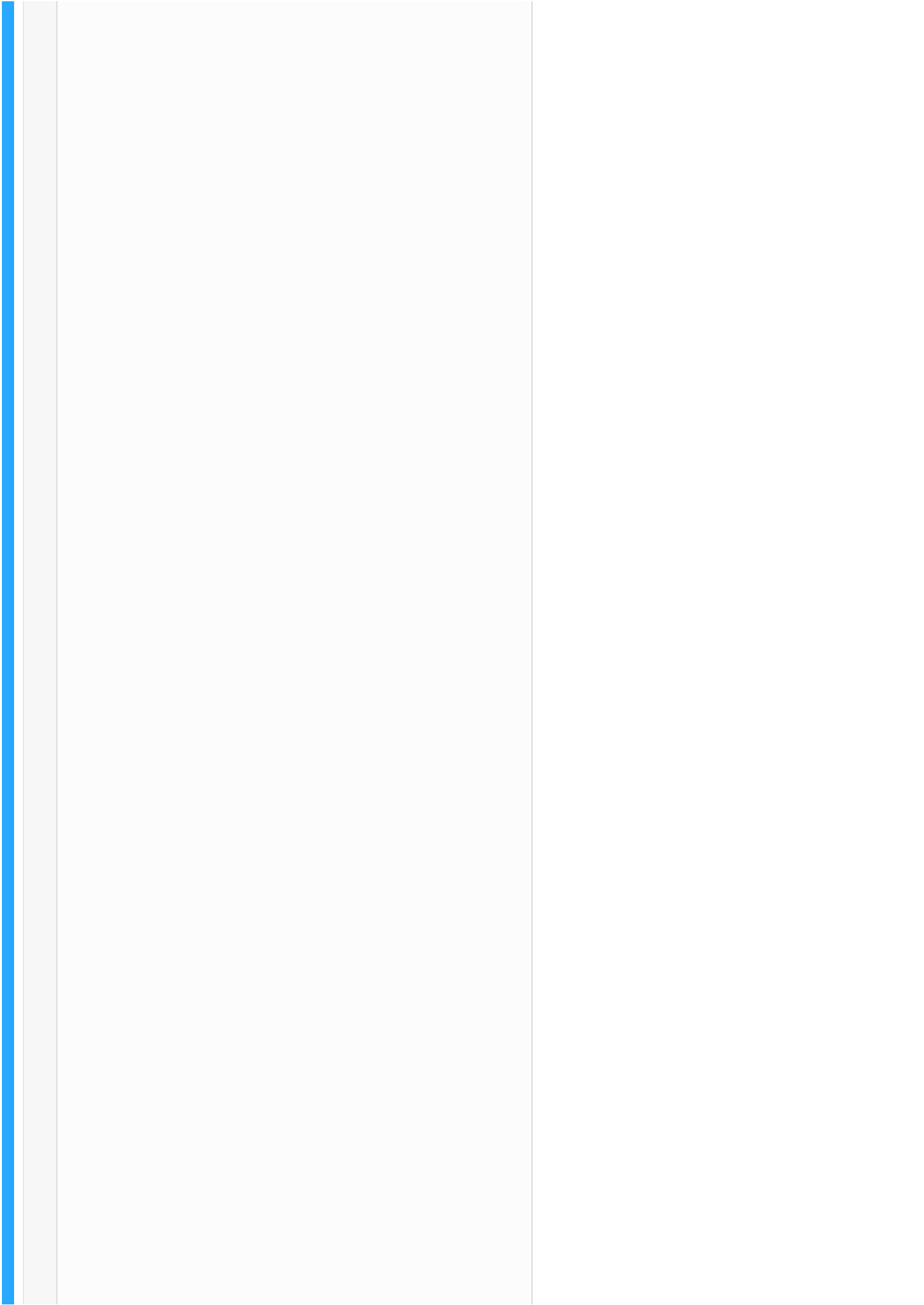


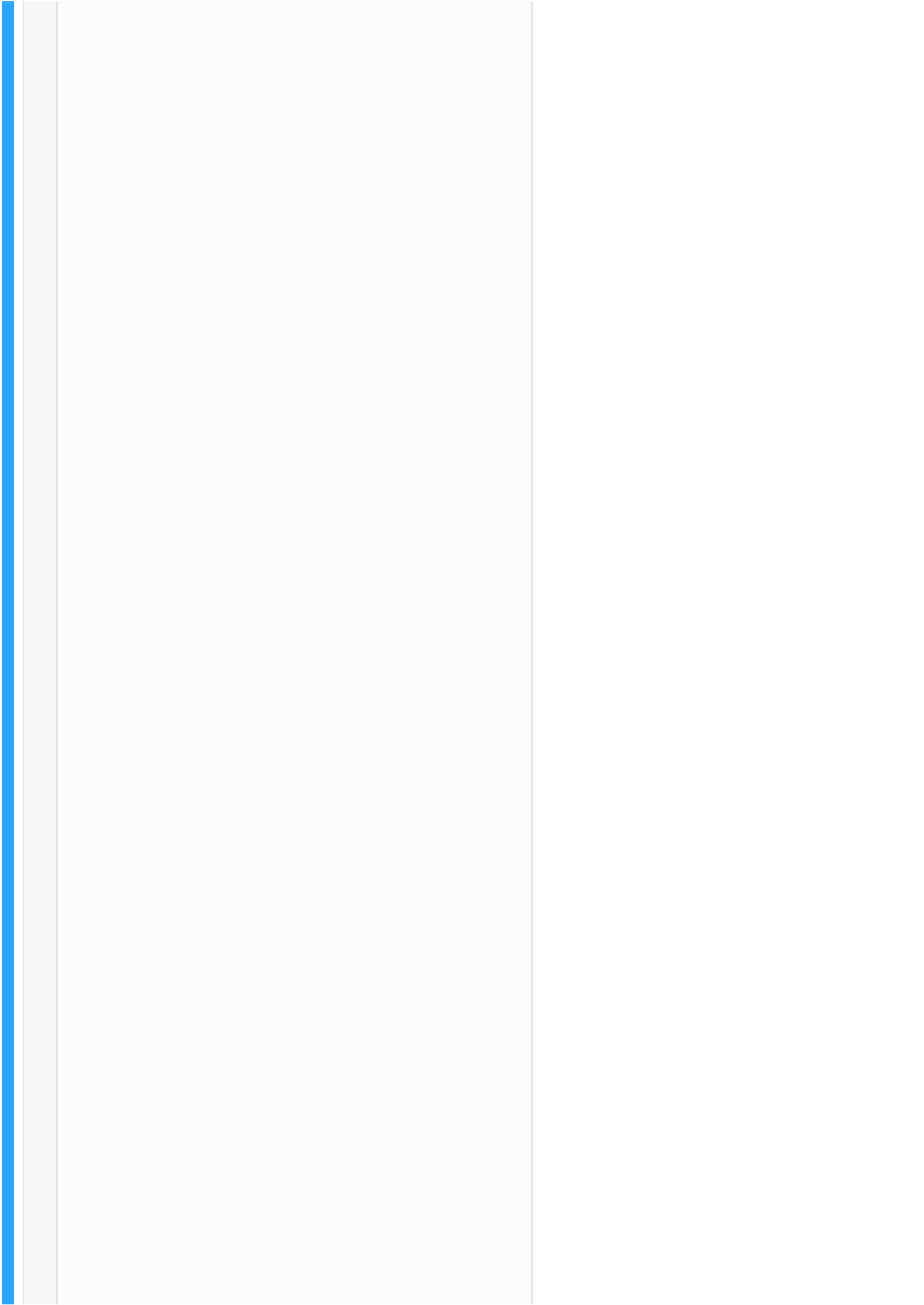






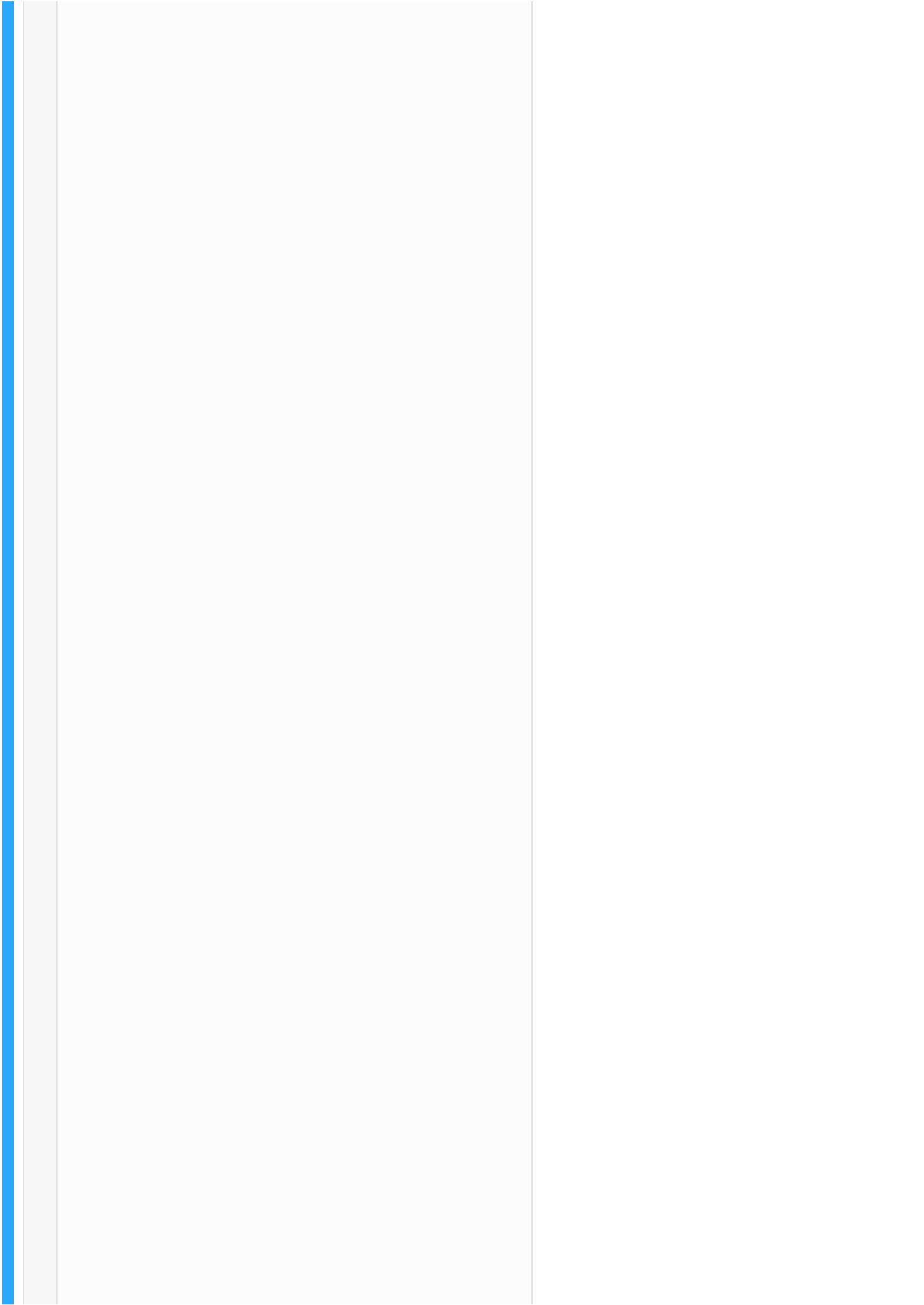


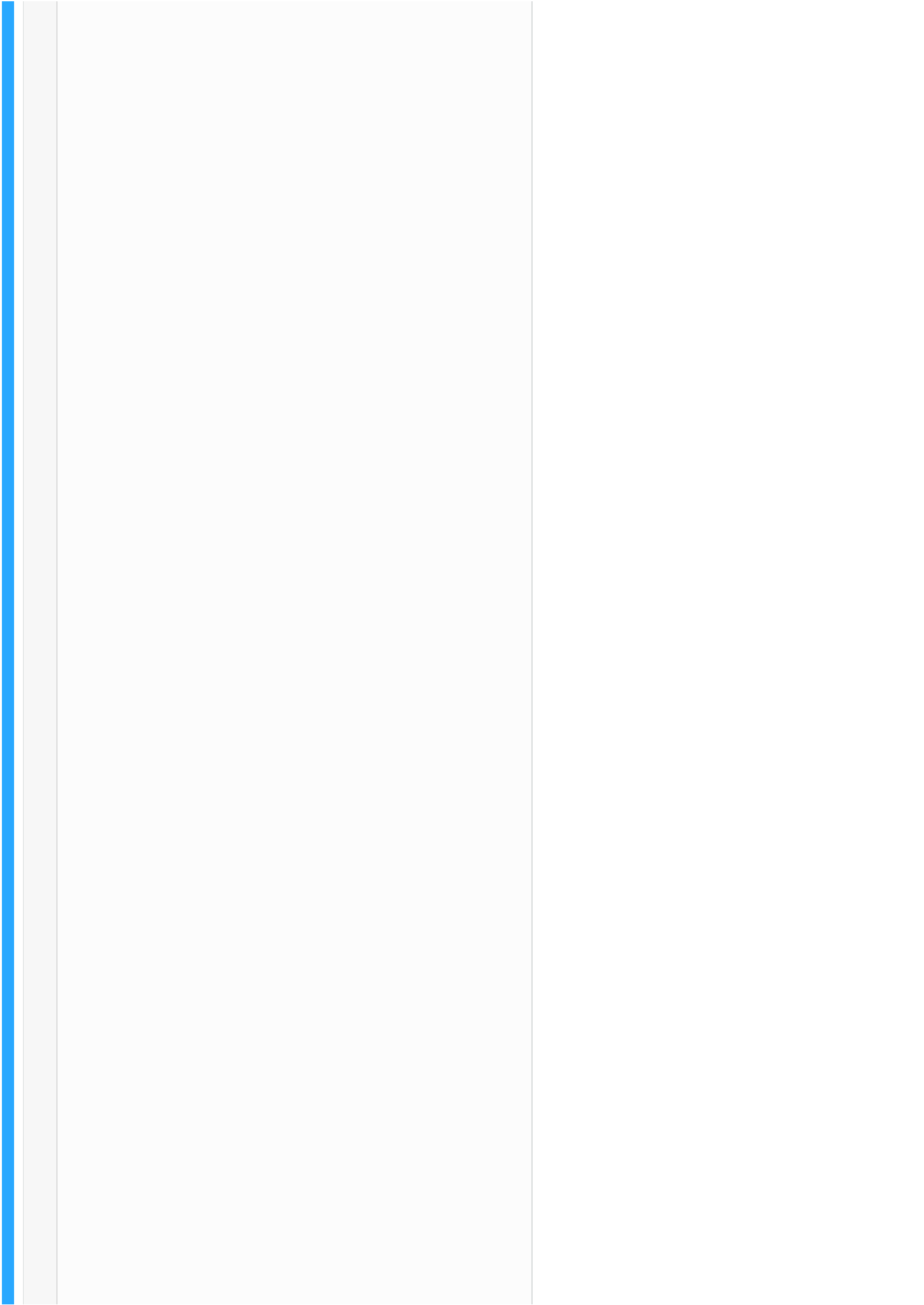










































DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)

This is one of those useful API endpoints that your users can use to check that everything is working as expected.

The Action Under Test

The code for this endpoint is a class called `EchoAction` which looks like this:

```
1 namespace App\Action;
2
3 use Psr\Http\Message\ServerRequestInterface as Request;
4 use Psr\Http\Message\ResponseInterface as Response;
5
6 class EchoAction
7 {
8     public function __invoke(Request $request, Response $response, $args = [])
9     {
10         return $response->withJson($request->getQueryParams());
11     }
12 }
```









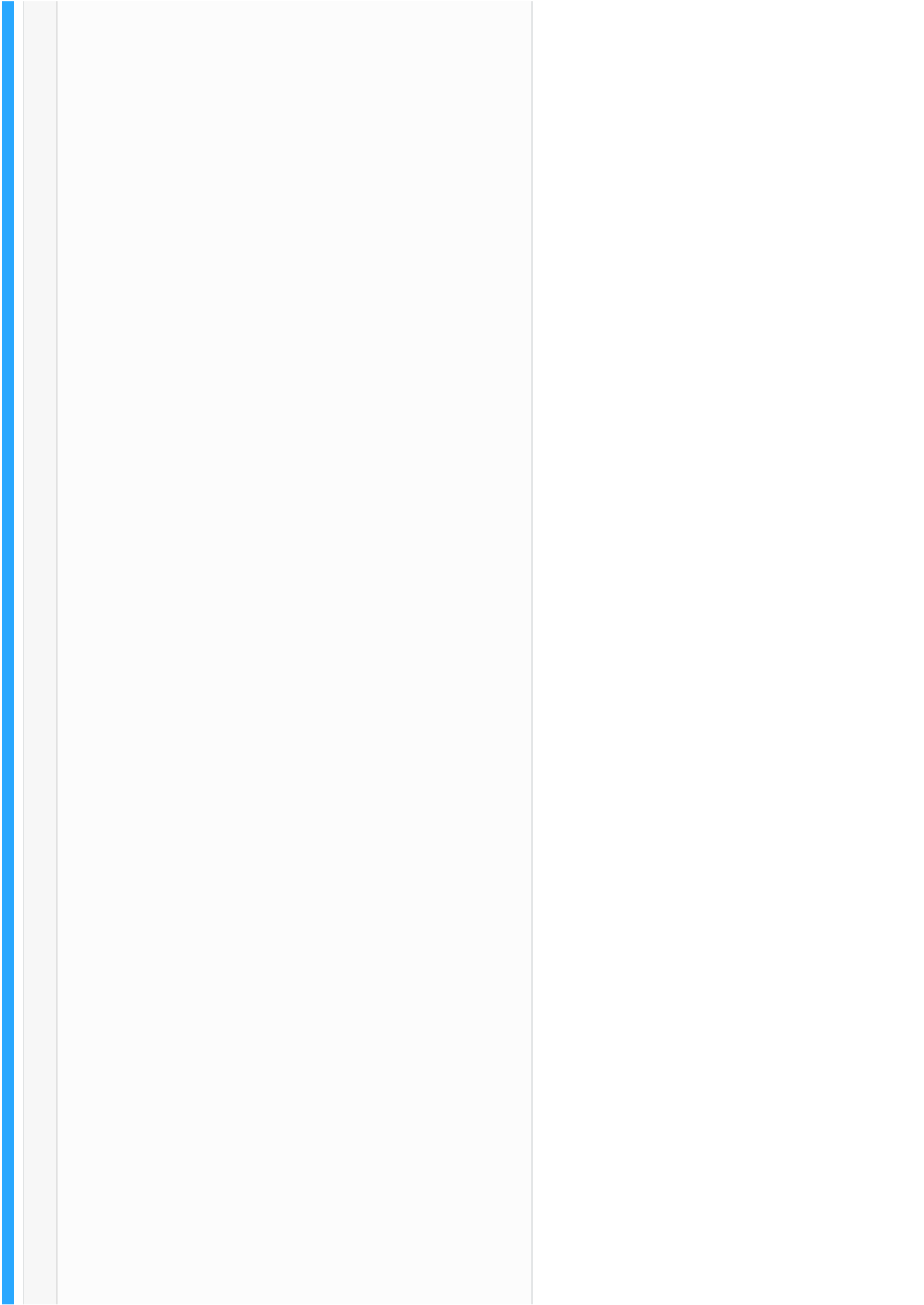


















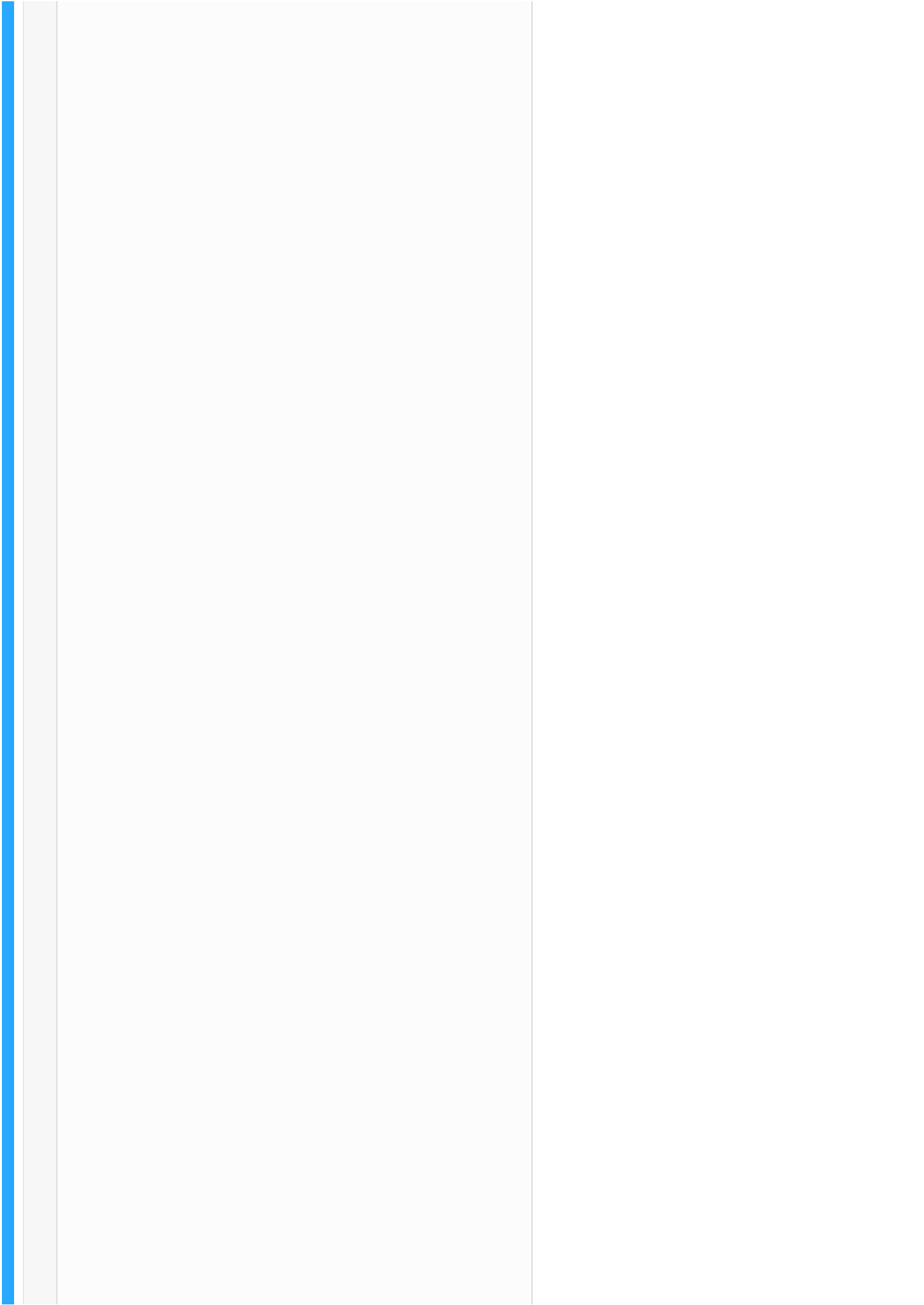


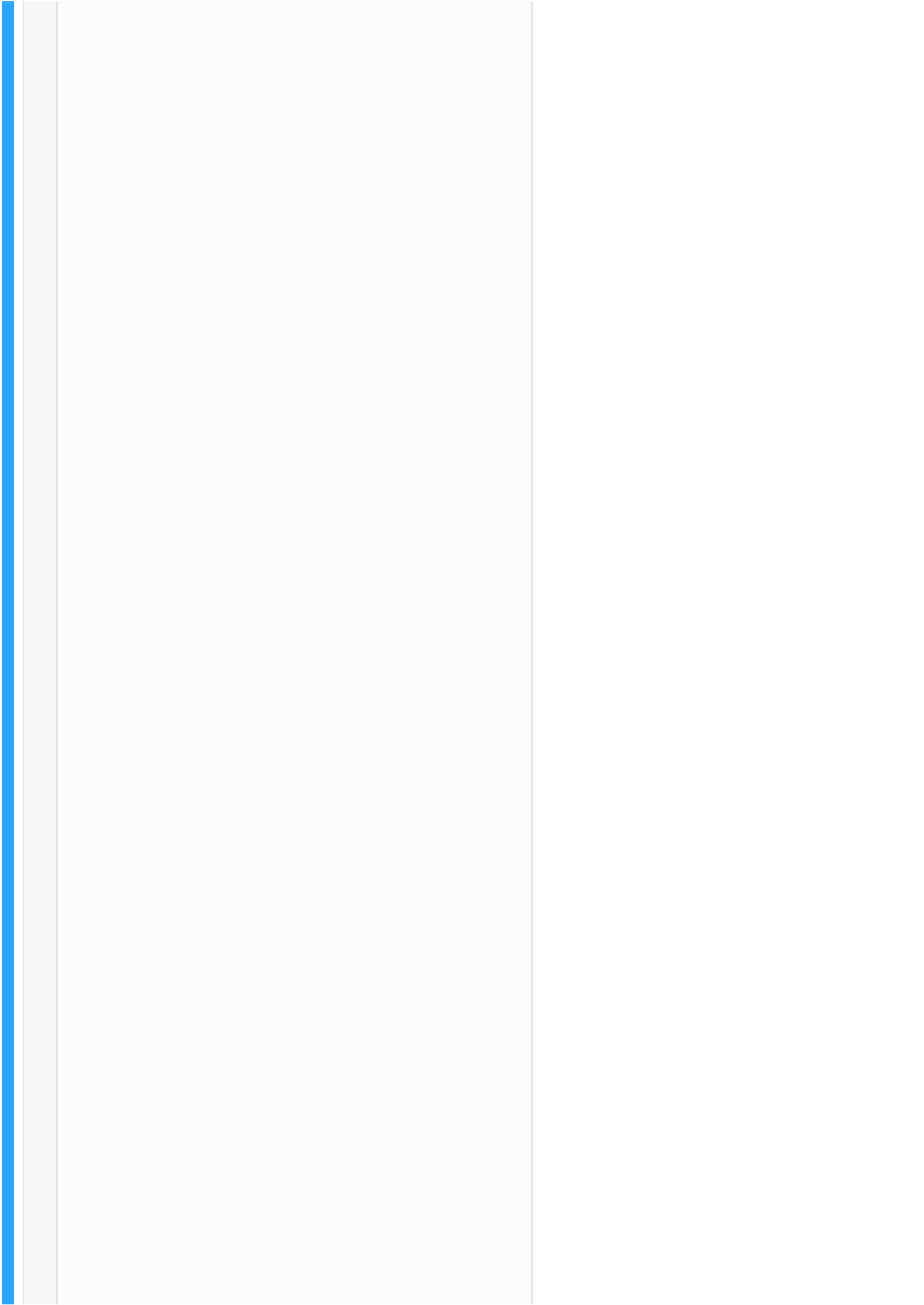


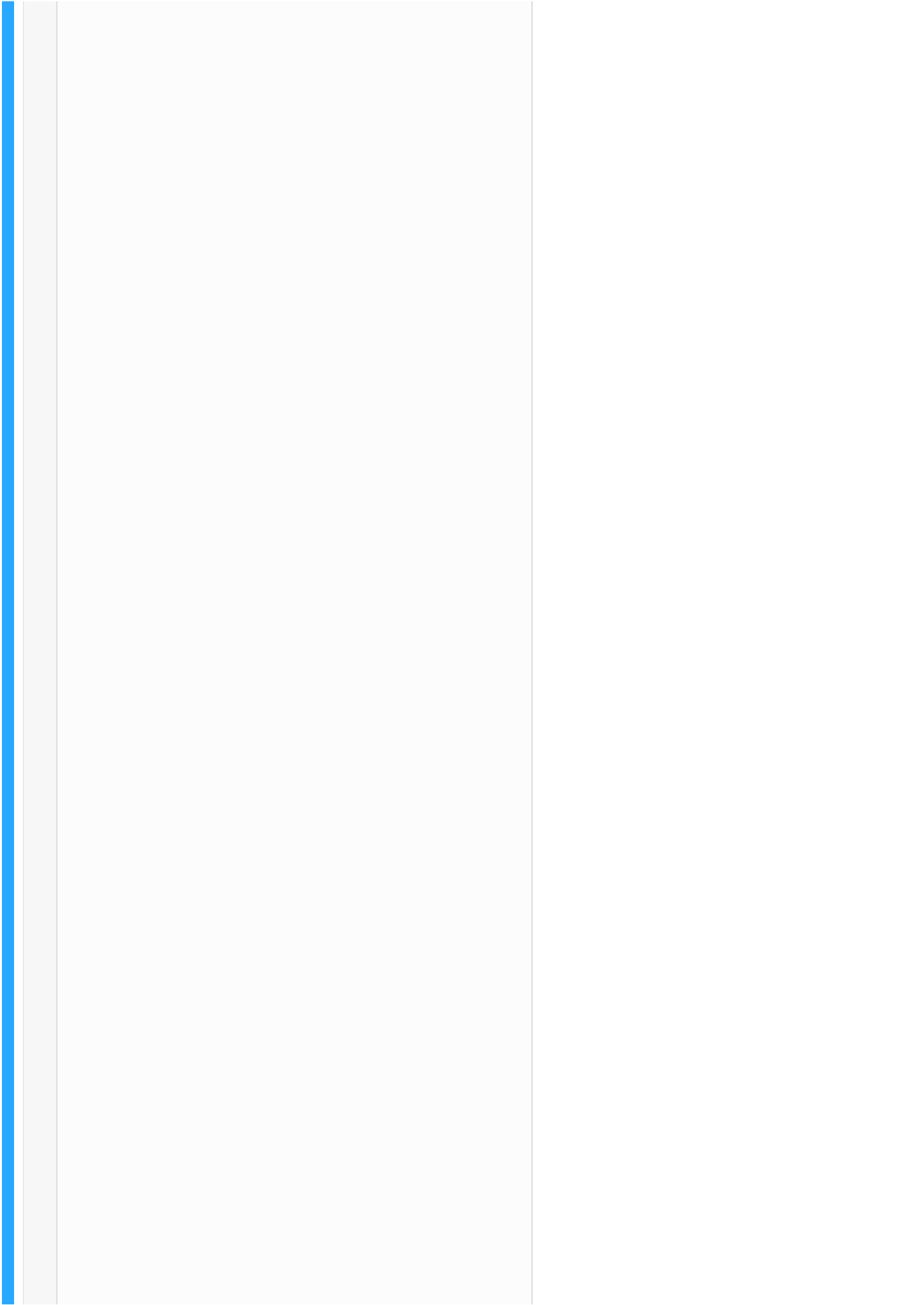






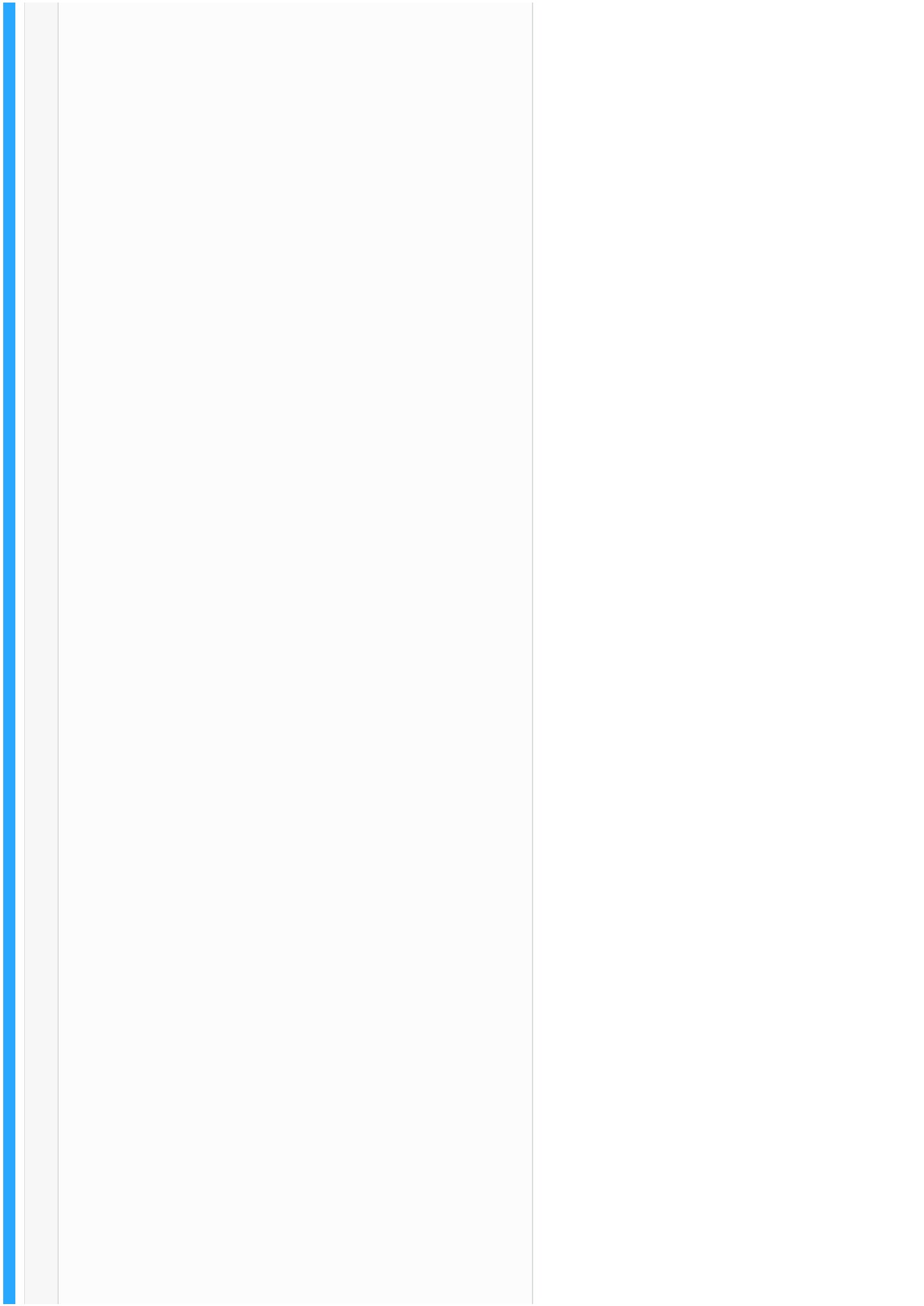


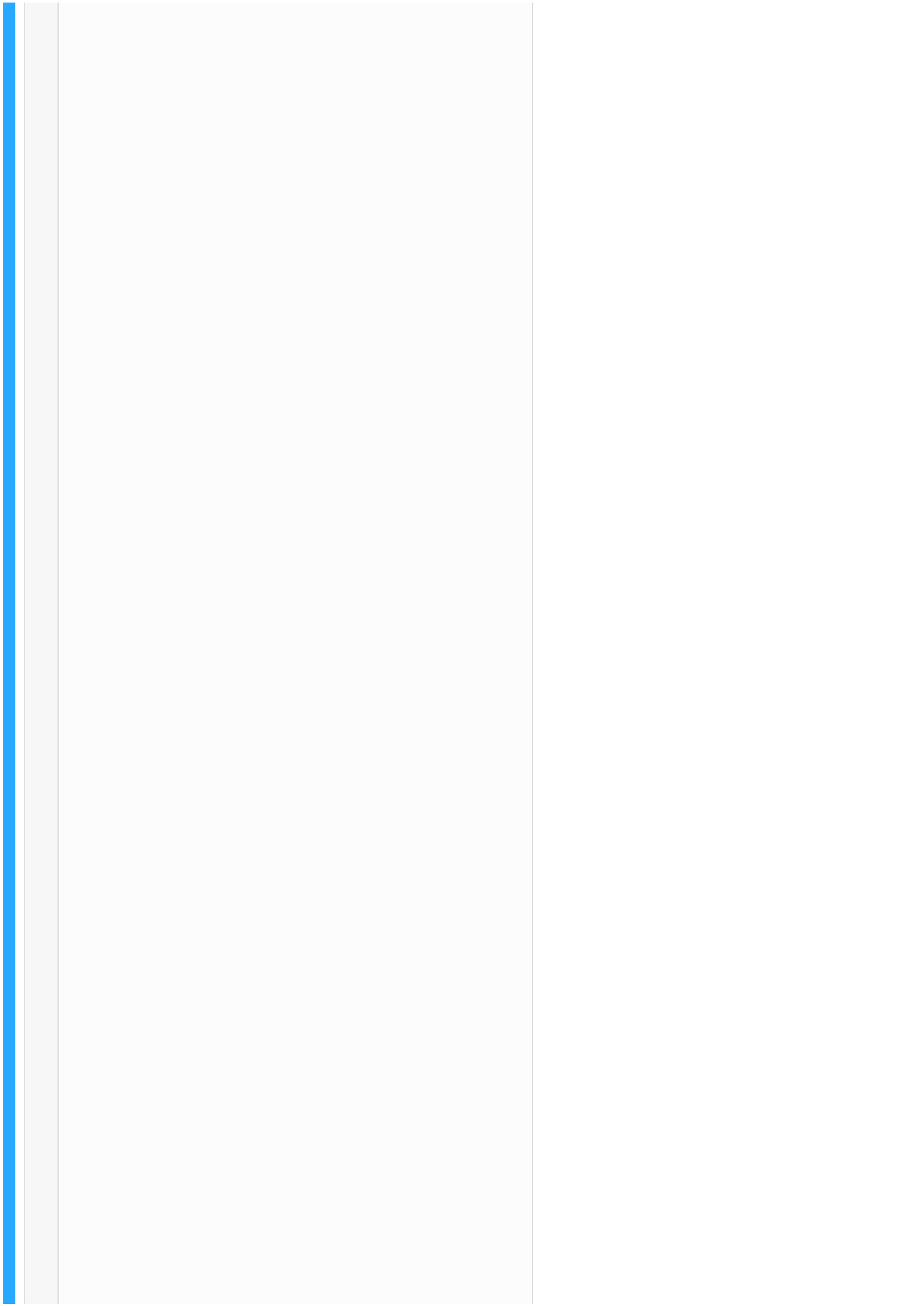


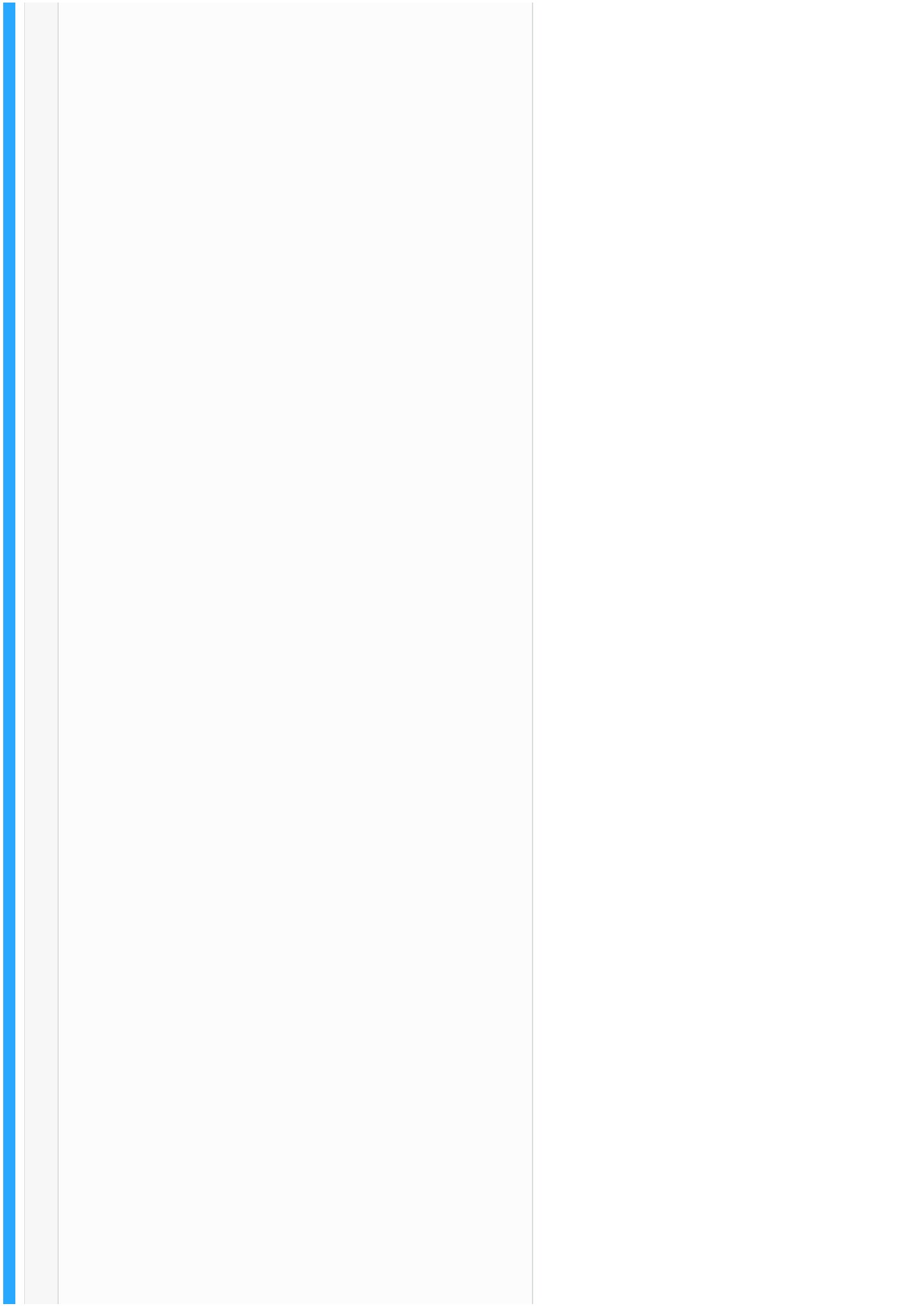














DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)

and it is registered with the Slim App like so:

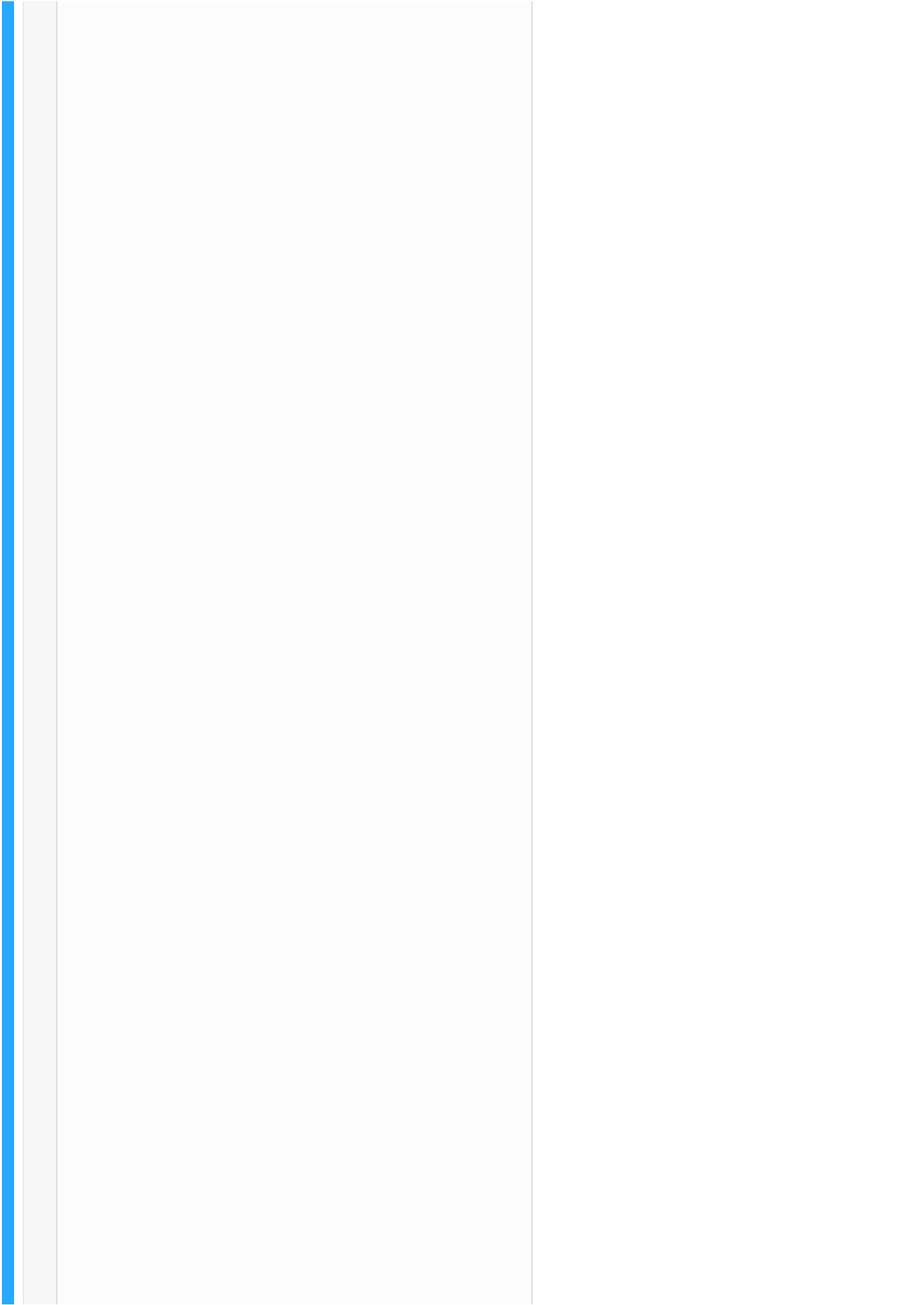
```
1 $app->get('/echo', App\Action\EchoAction::class)  
  ;
```

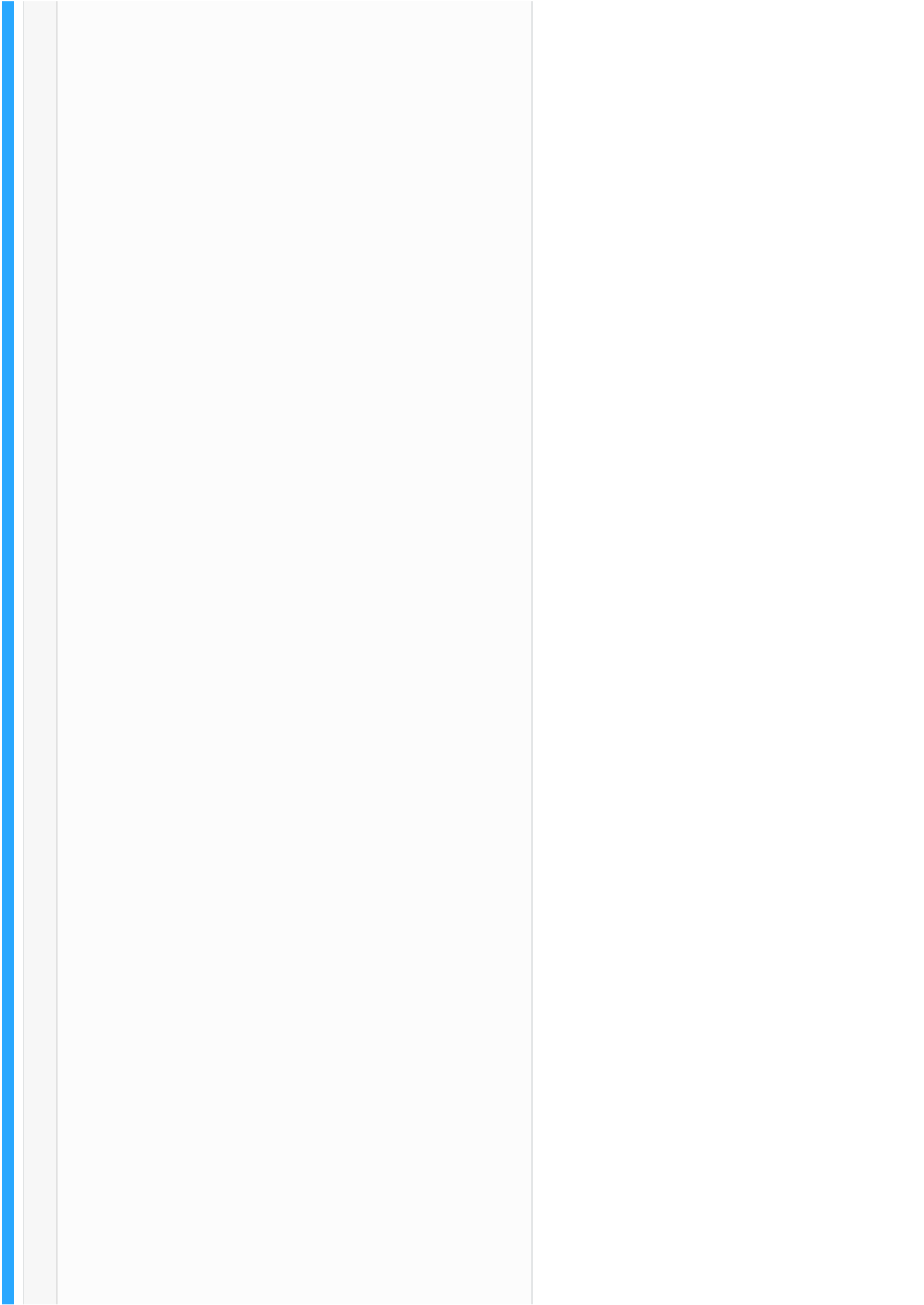






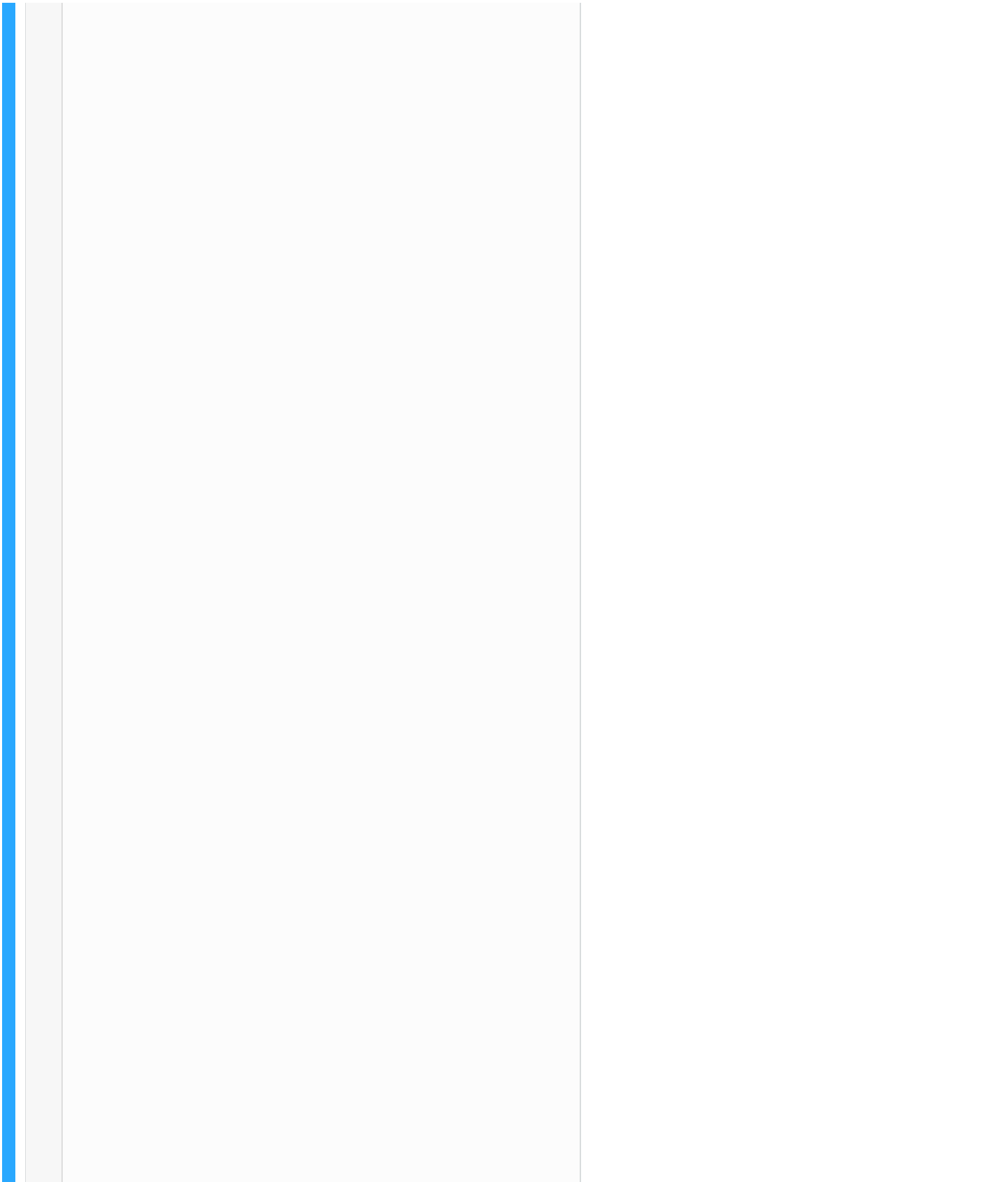


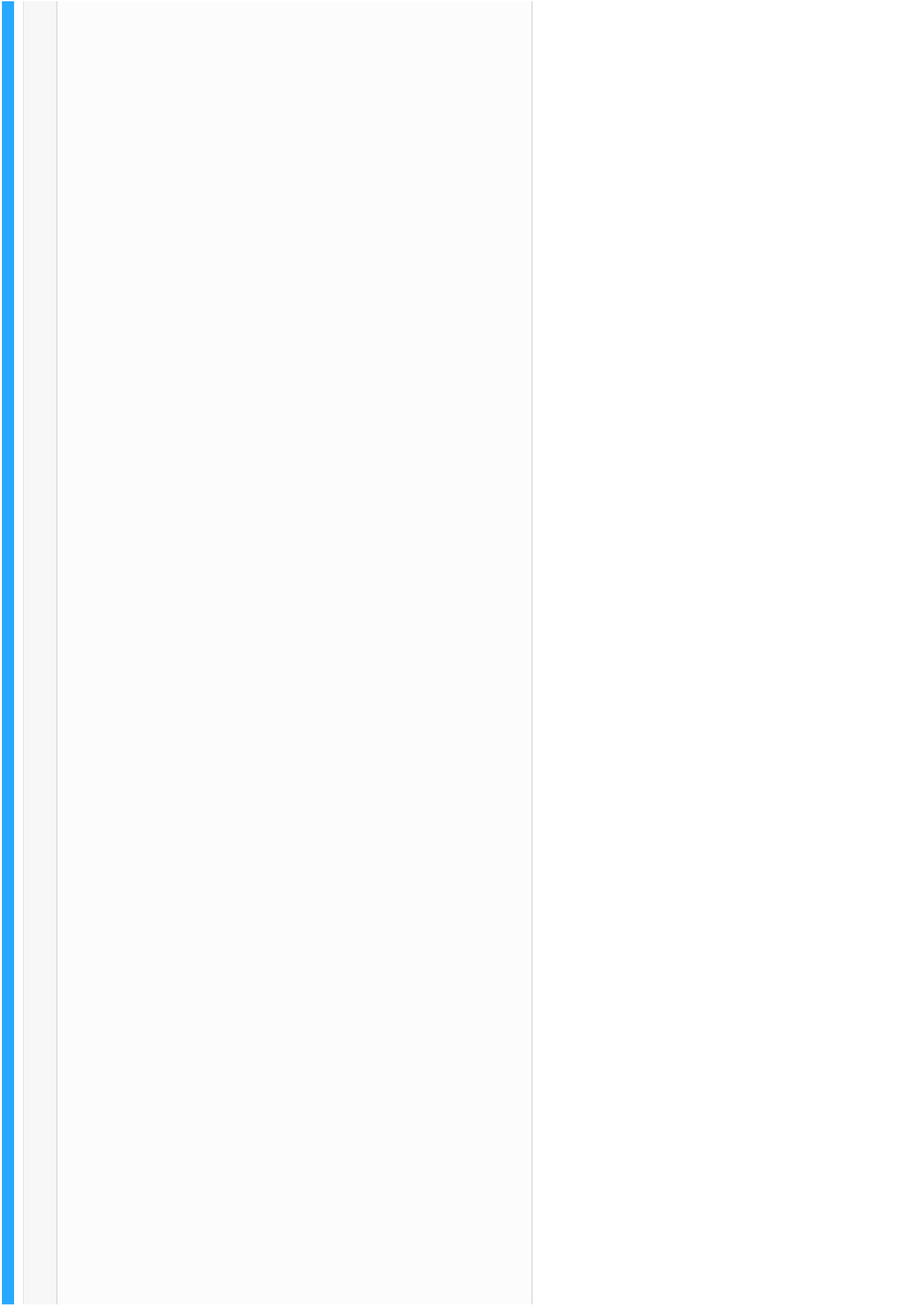


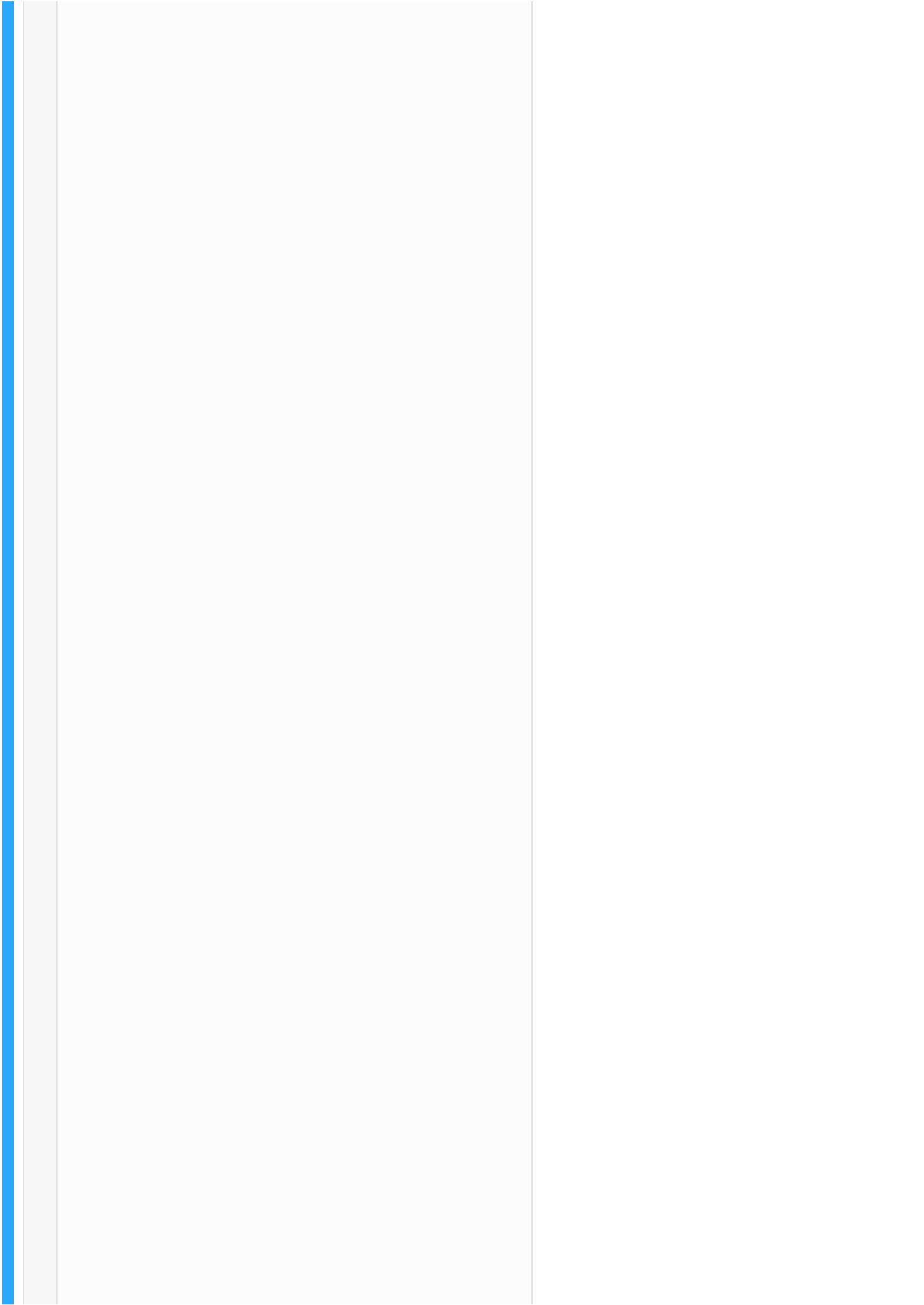








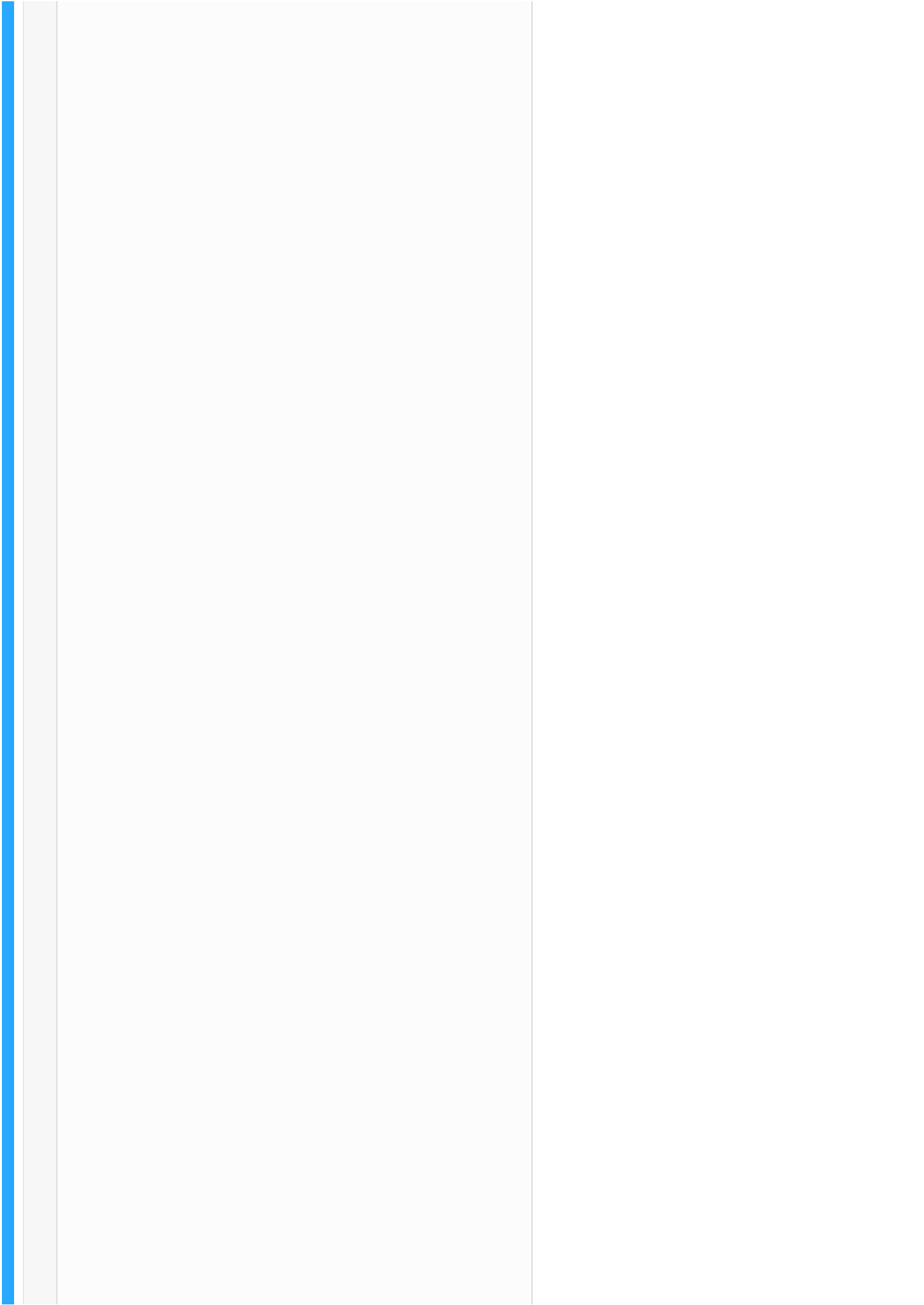
























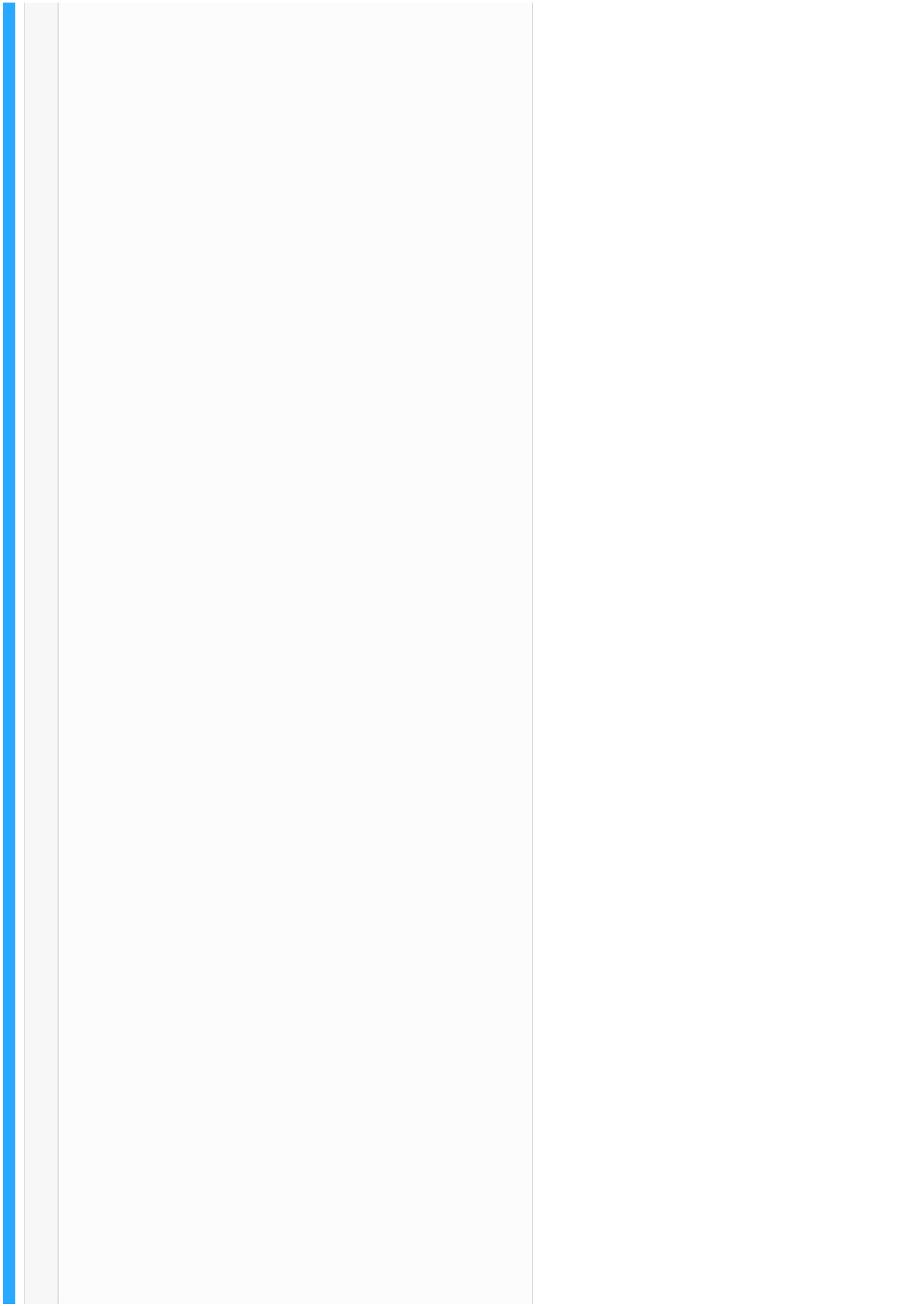














DZone Web Dev Zone

[Log In](#) / [Sign Up](#) 



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)

Testing

Testing it isn't too complex as there are no dependencies on the EchoAction class itself, so we just have to instantiate the class, invoke it, and write a test.

For a URL of the form `/echo?foo=bar`, the core test code is:

```
1 $action = new \App\Action\EchoAction();
2 $response = $action($request, $response);
3 $this->assertSame((string)$response->getBody(),
    json_encode(['foo' => 'bar']));
```







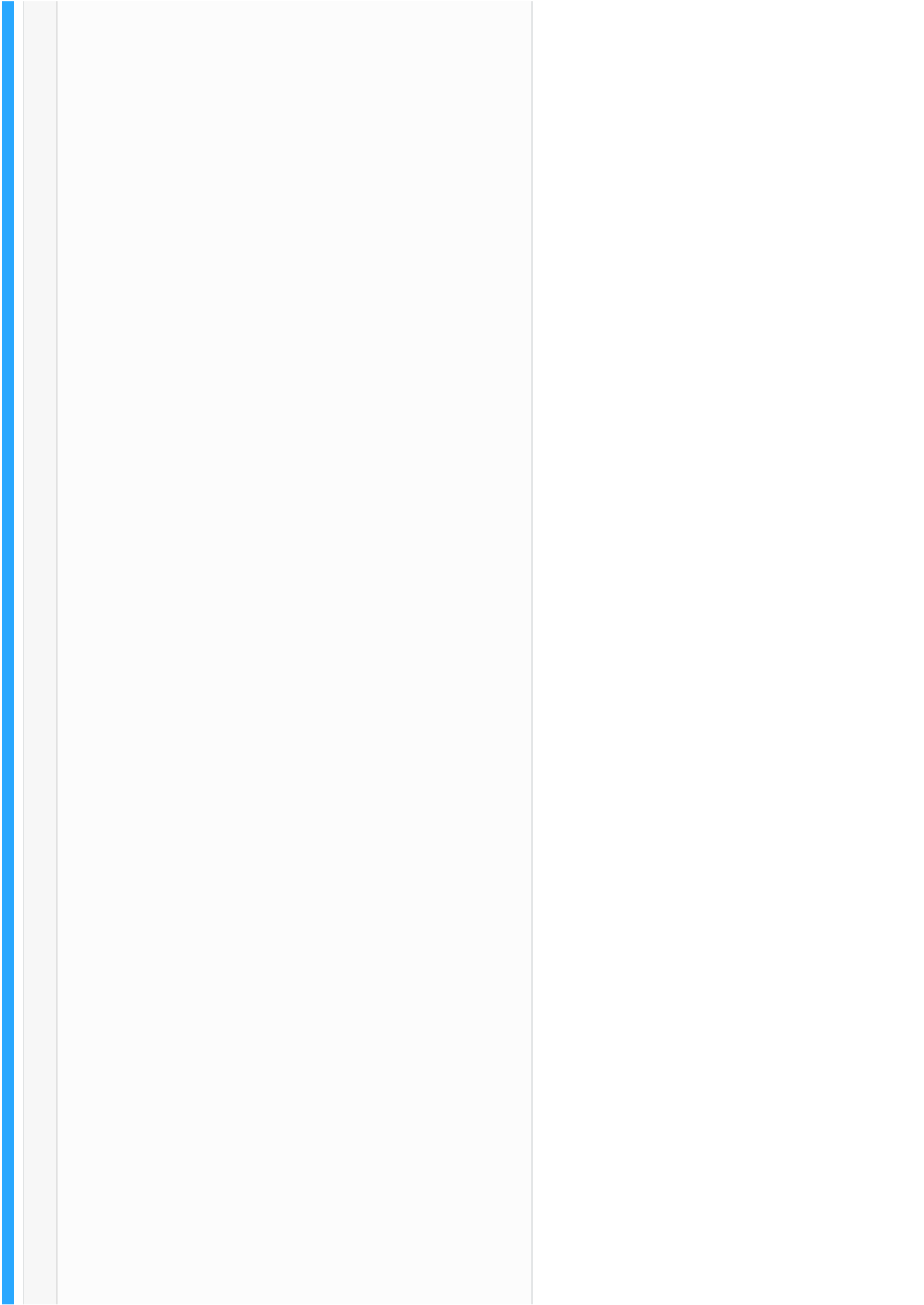


















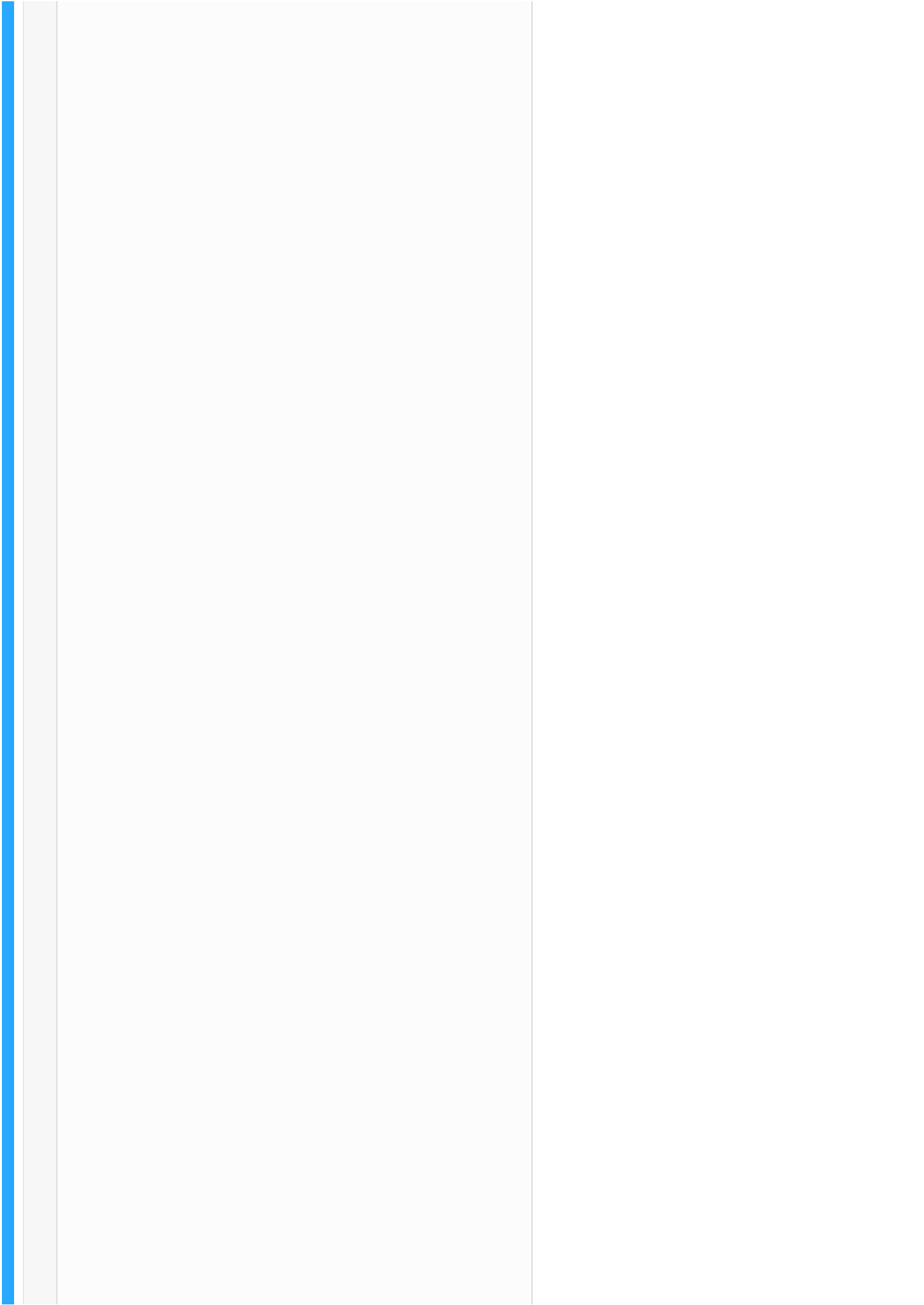


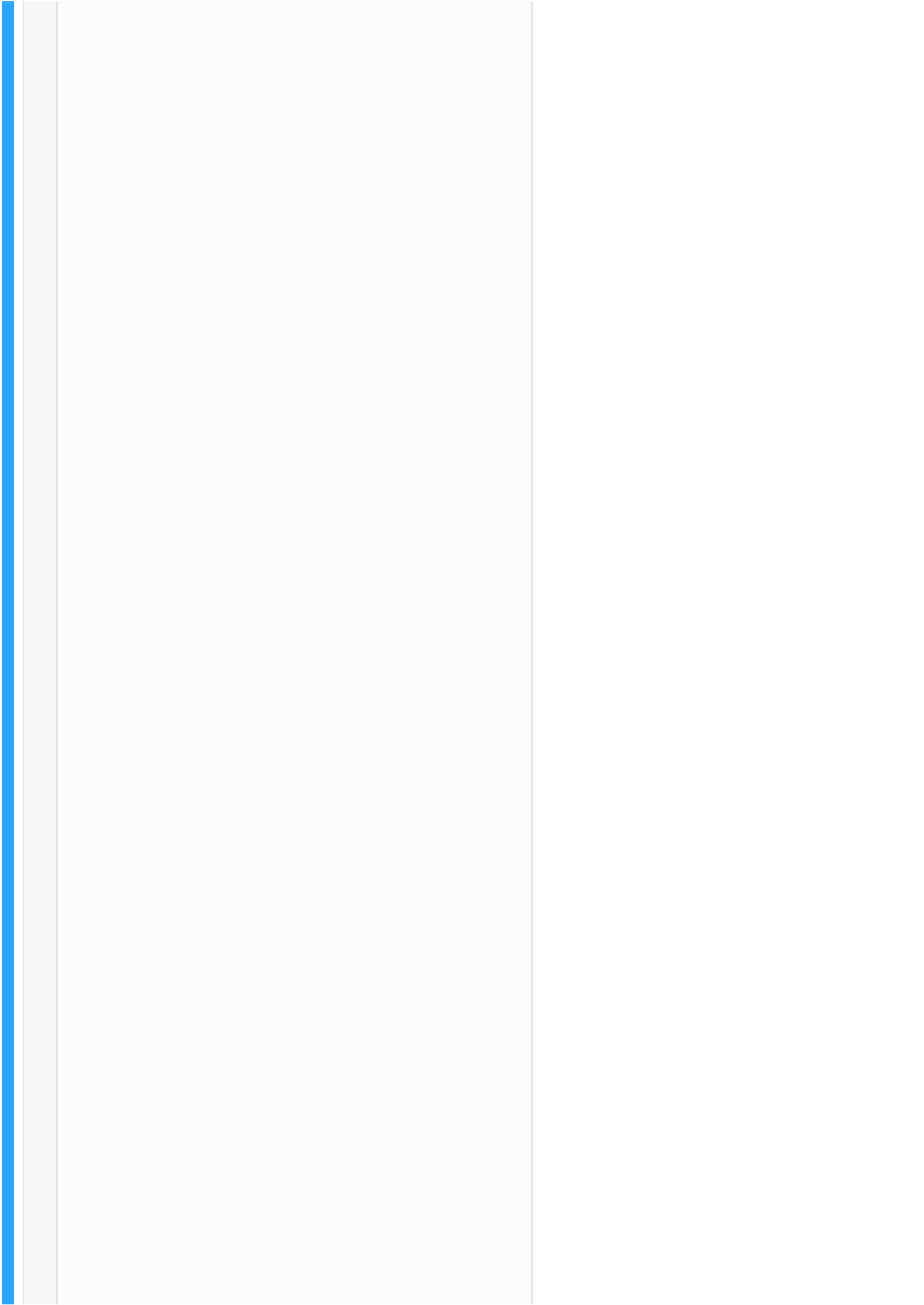


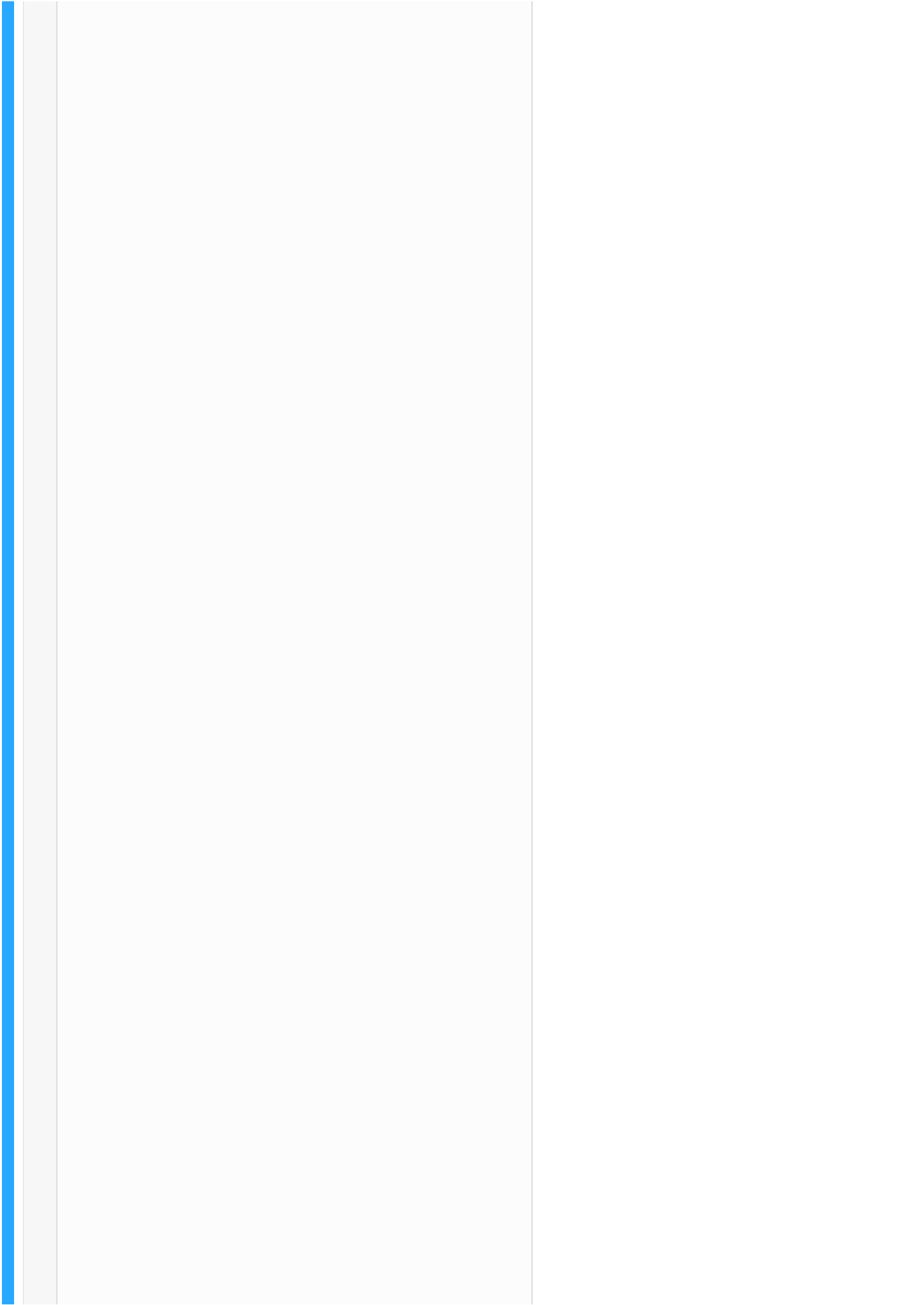








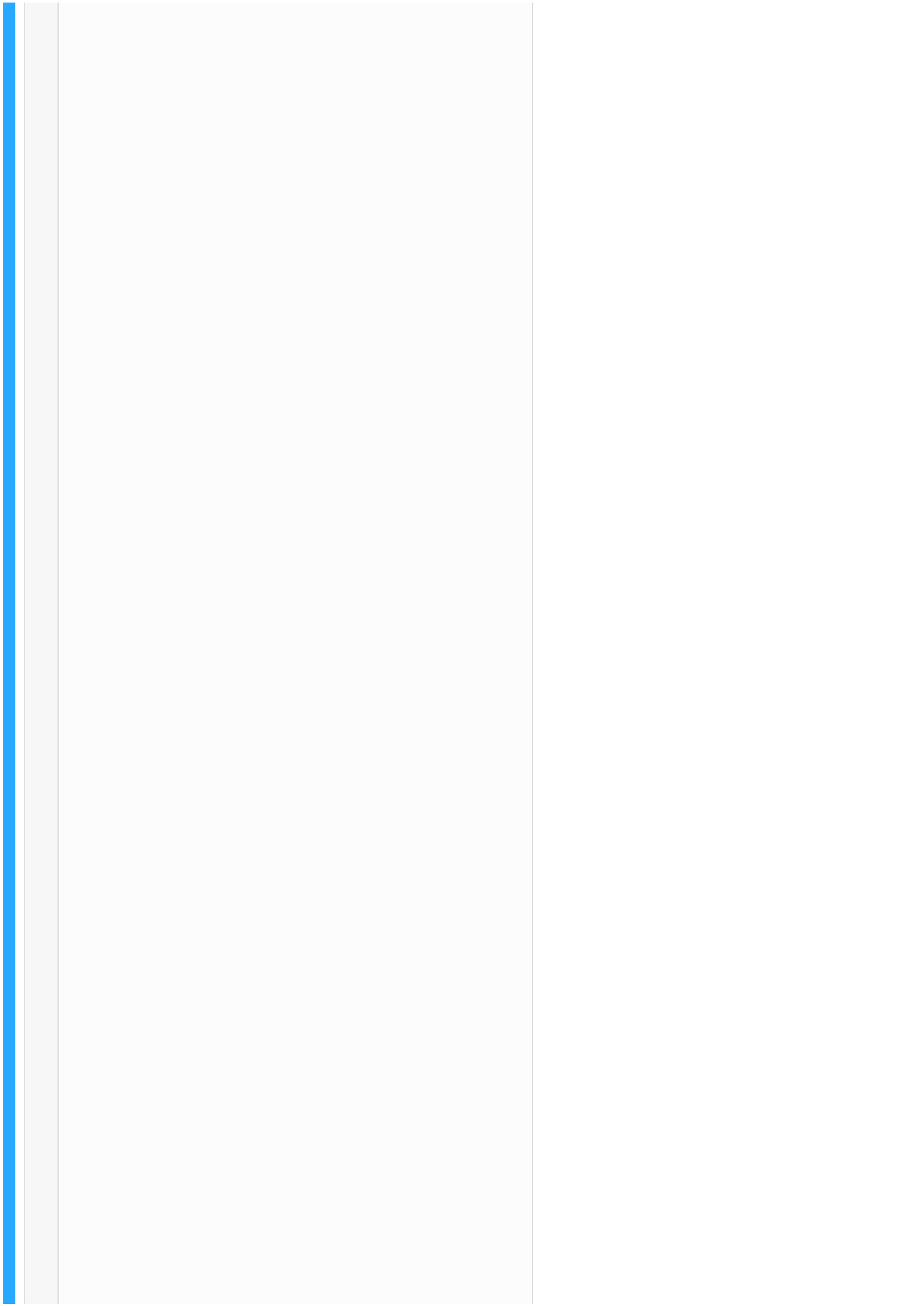


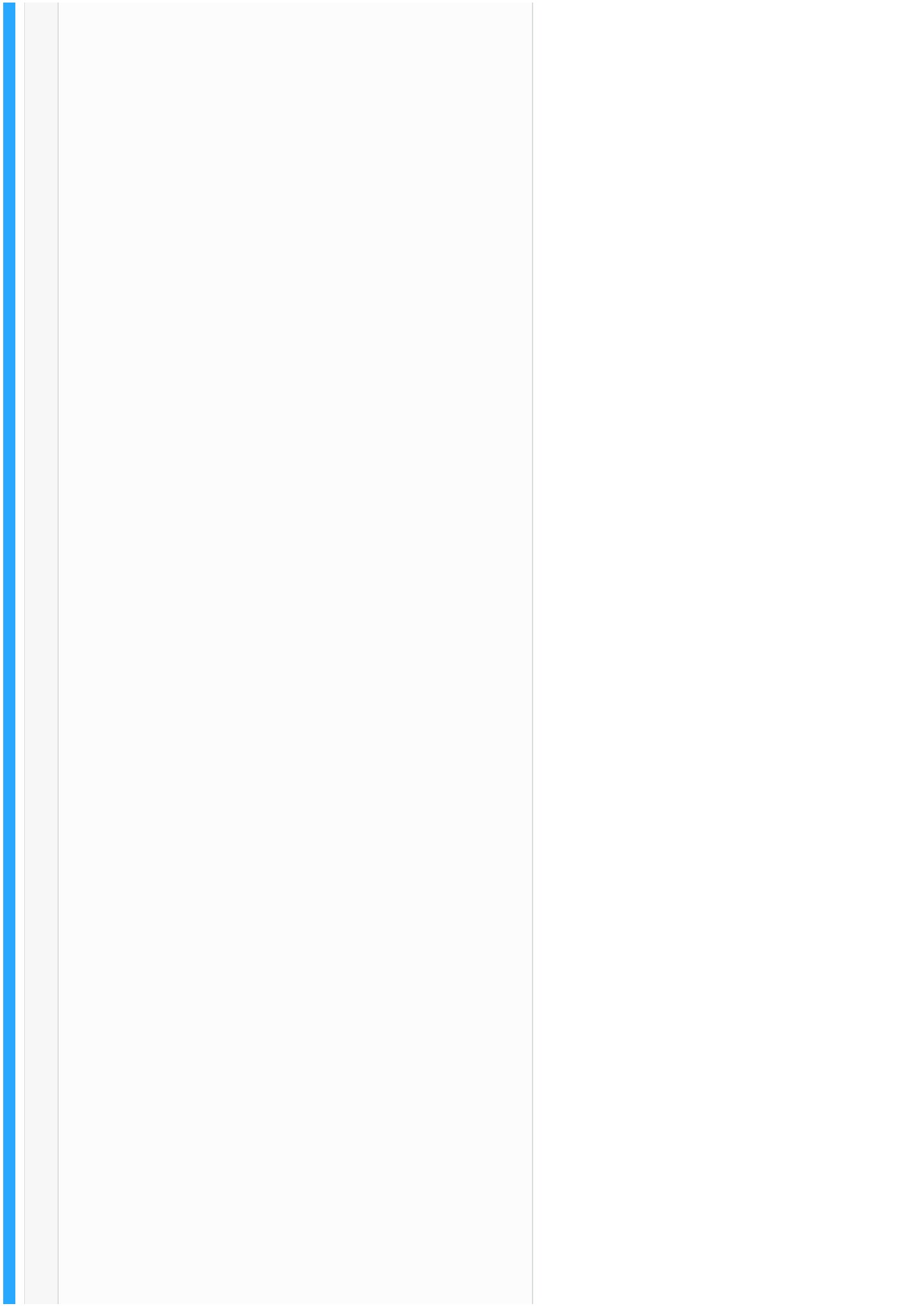














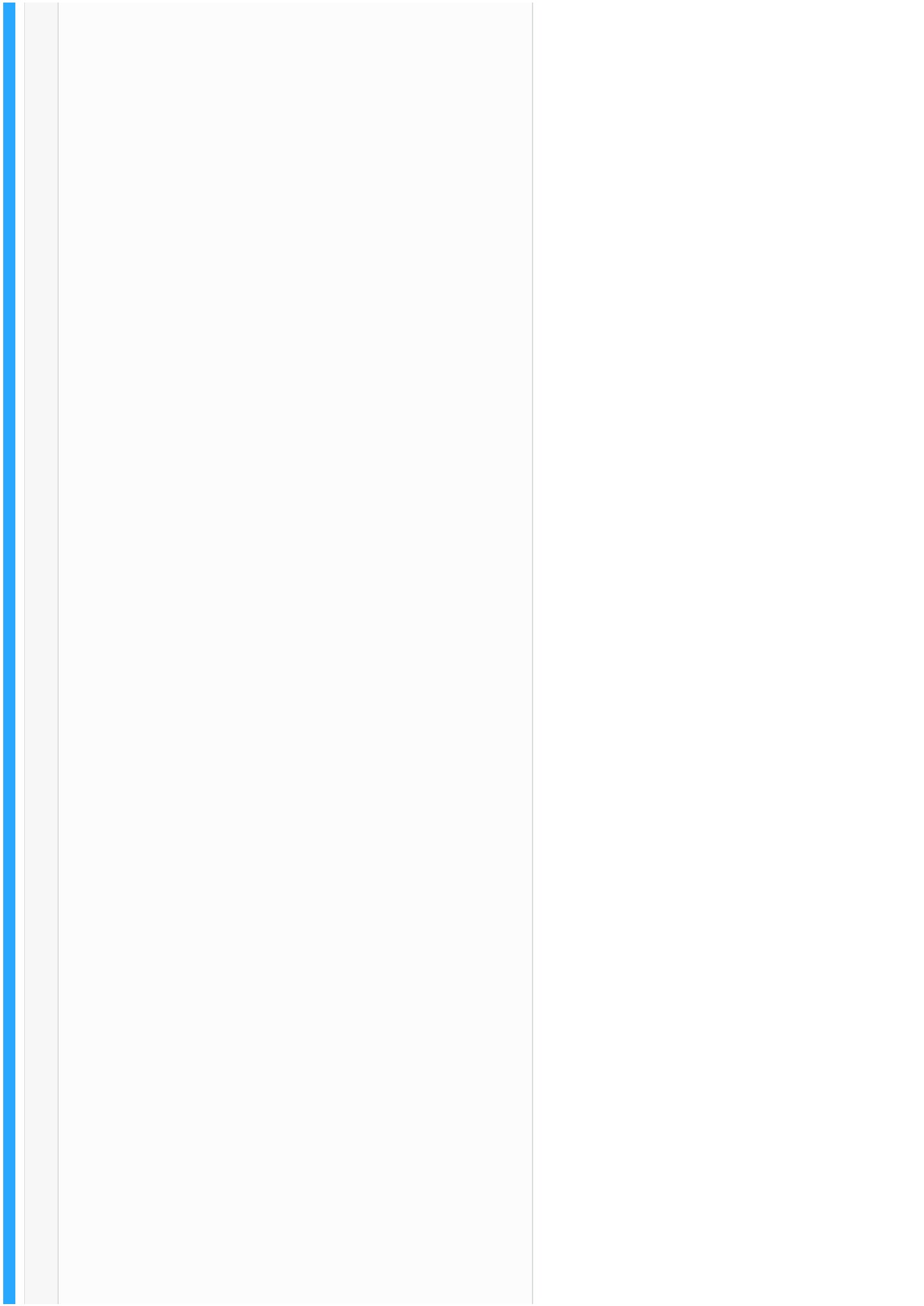
Creating the Request and Response

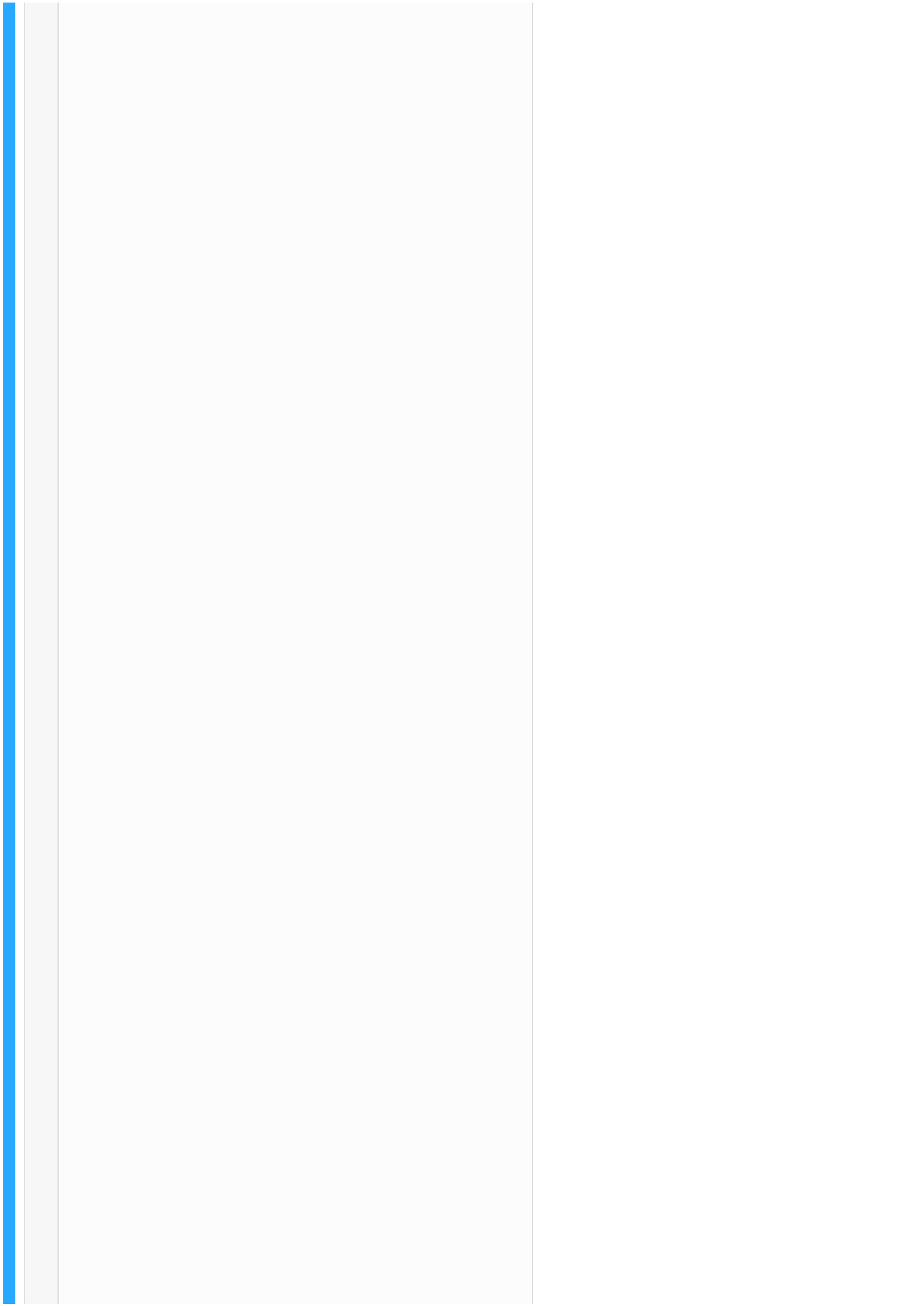
Creating the \$response is easy as the constructor parameters all have defaults:

```
1 $response = new \Slim\Http\Response();
```

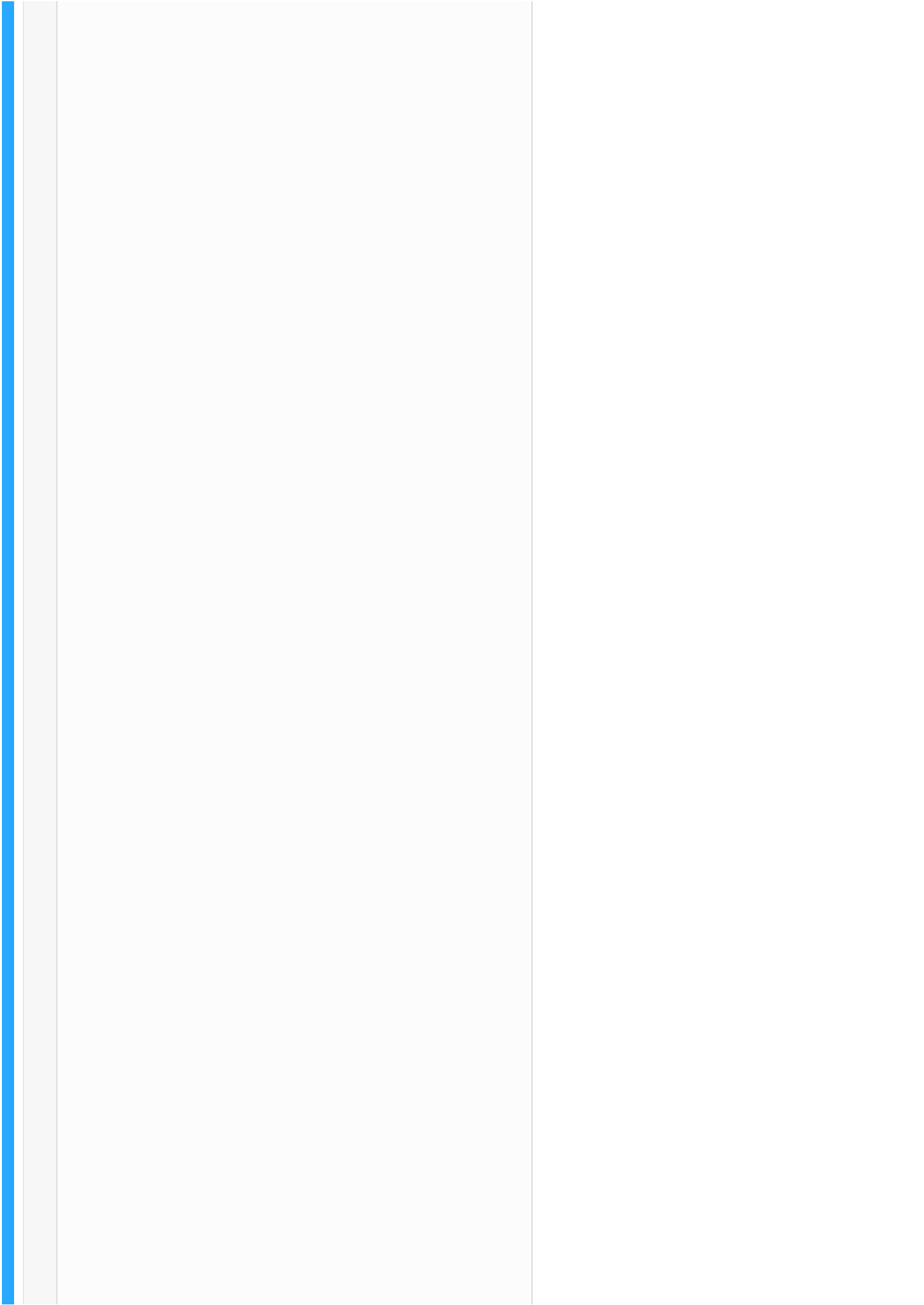


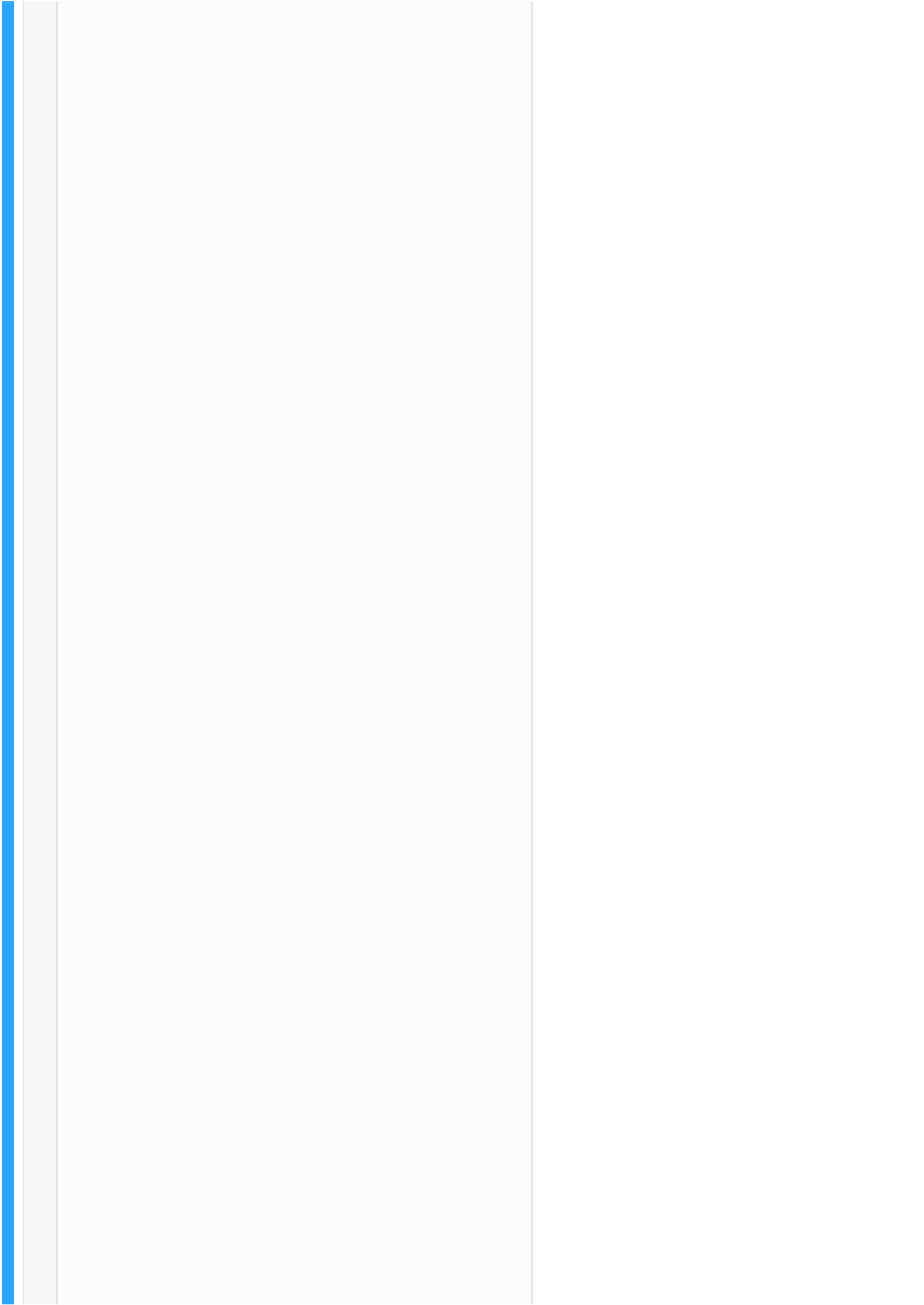






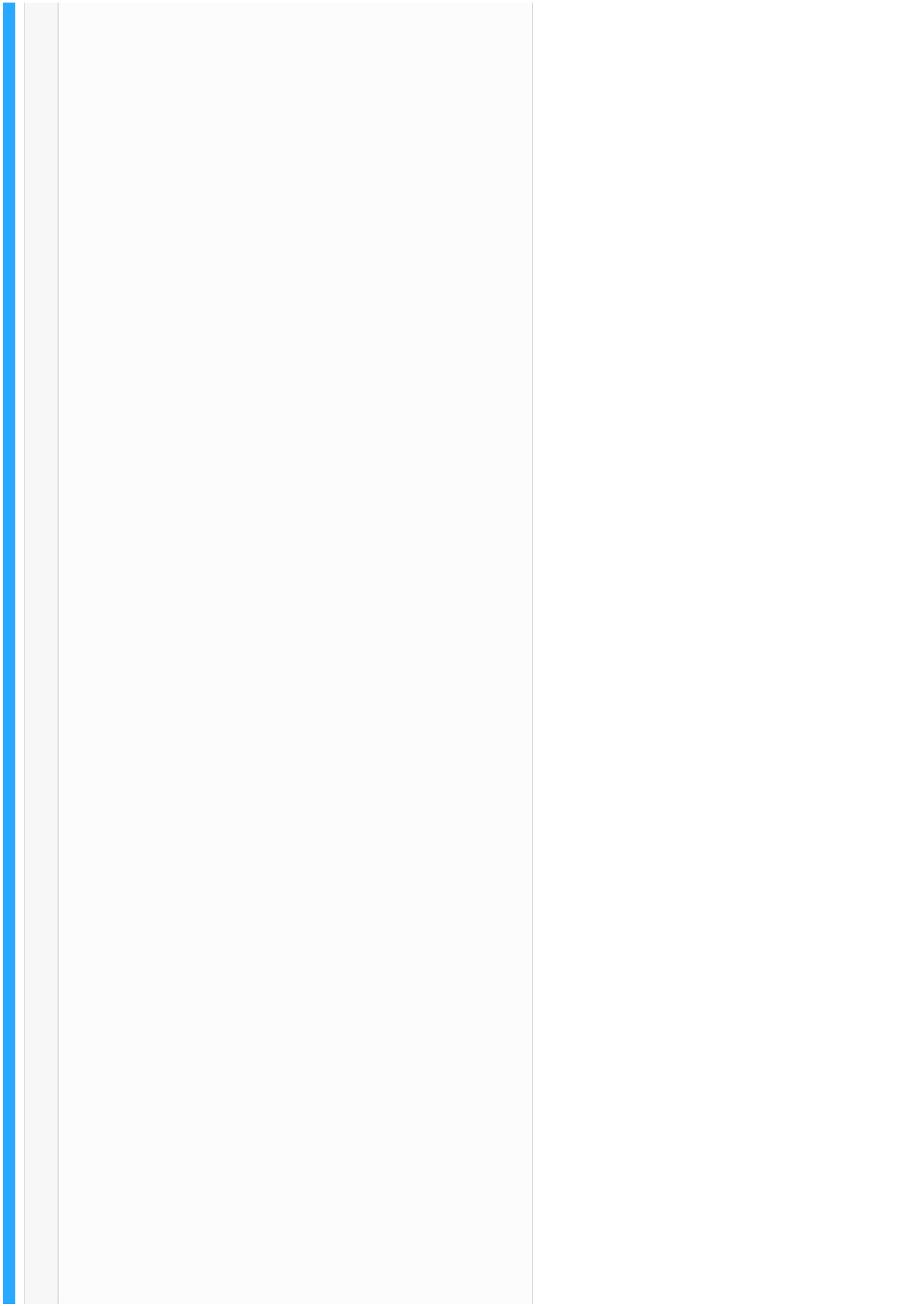


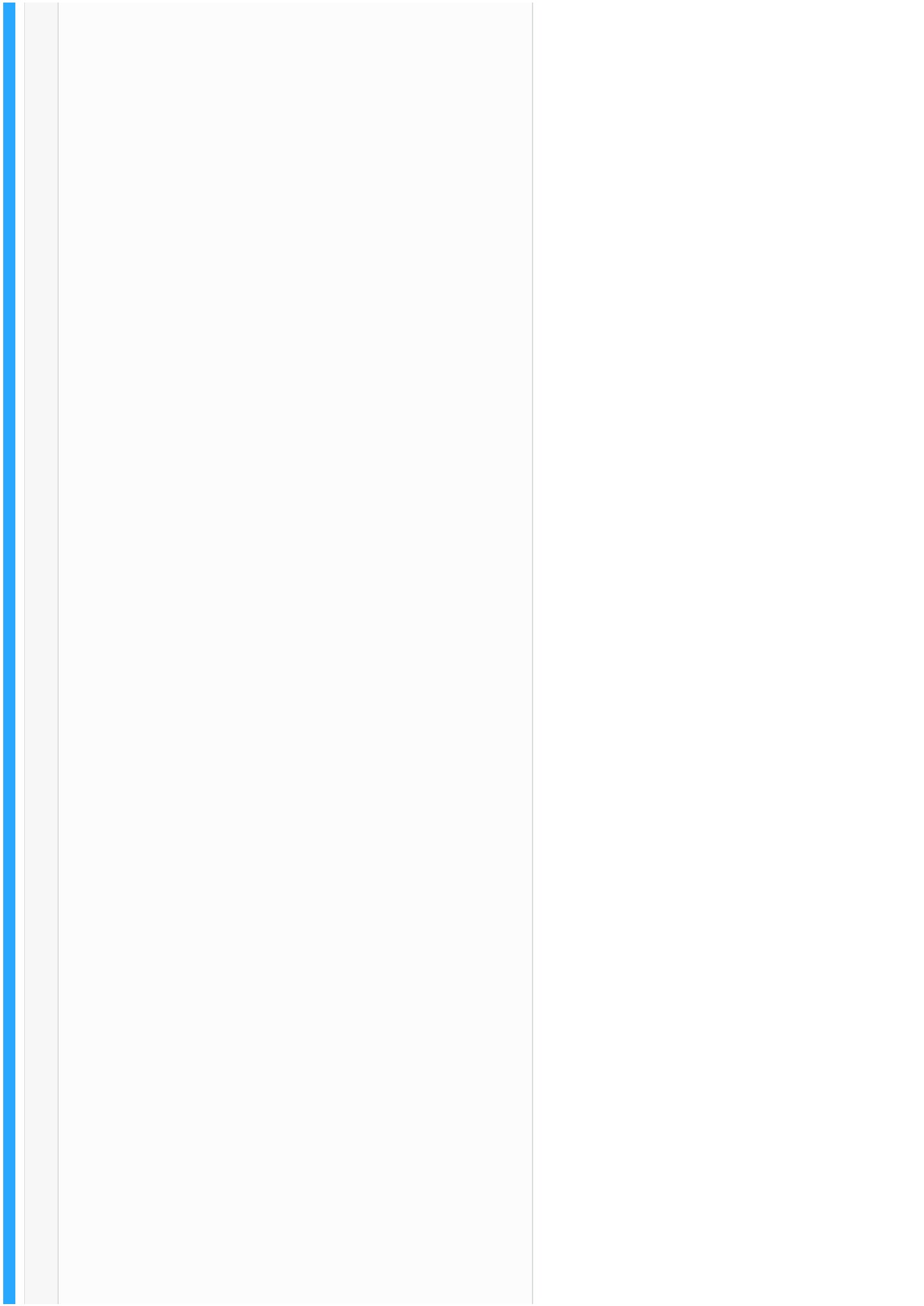












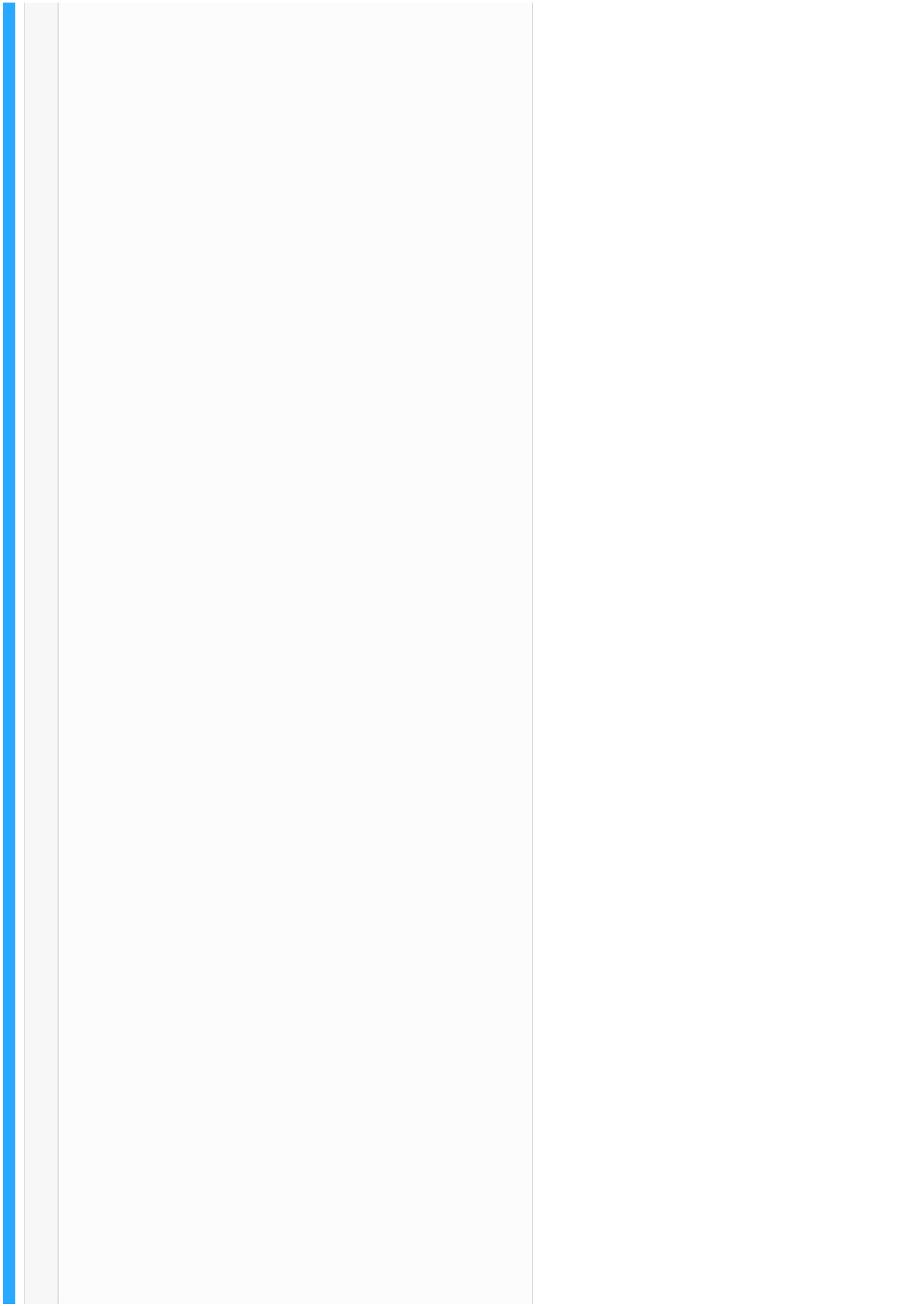


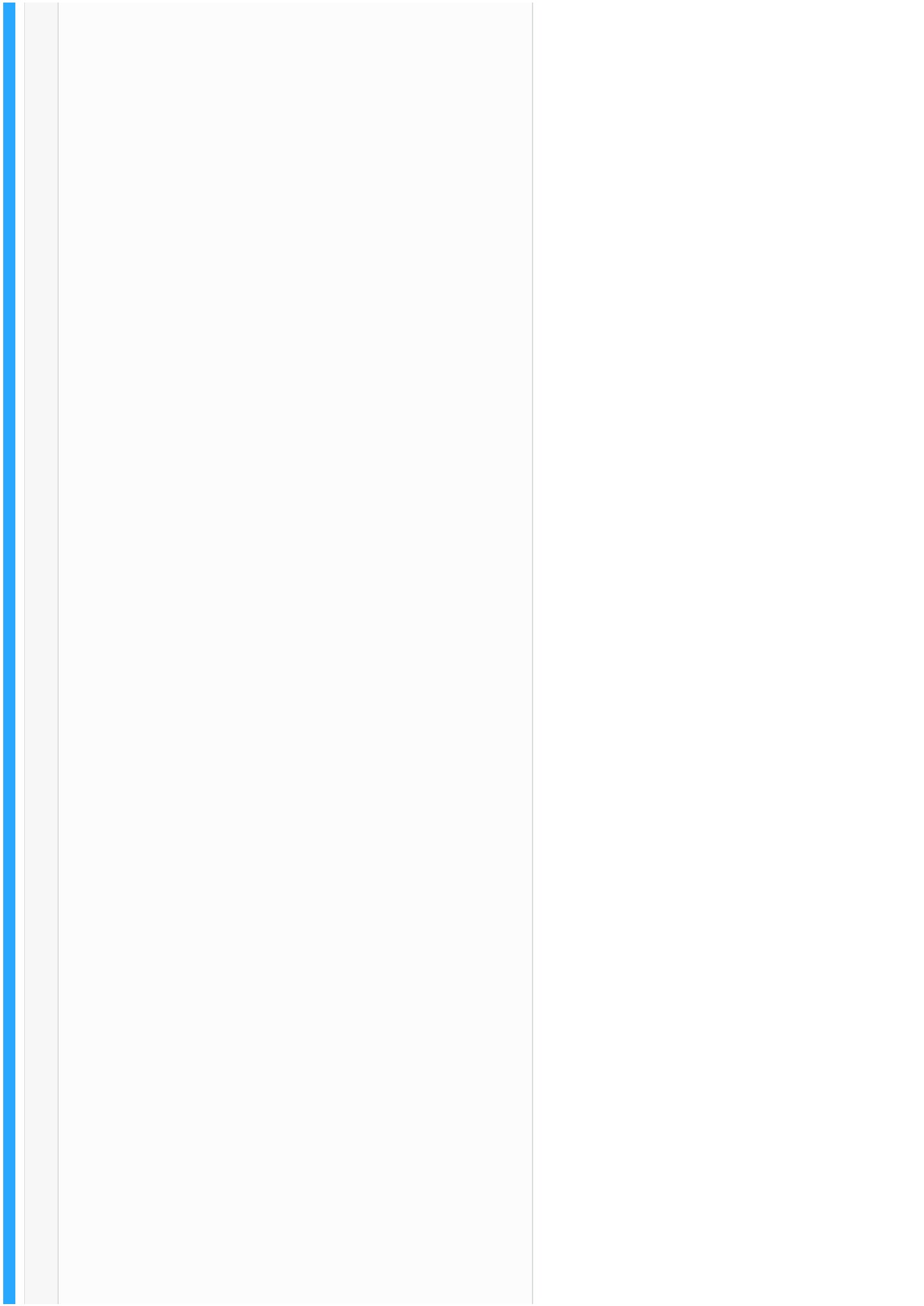








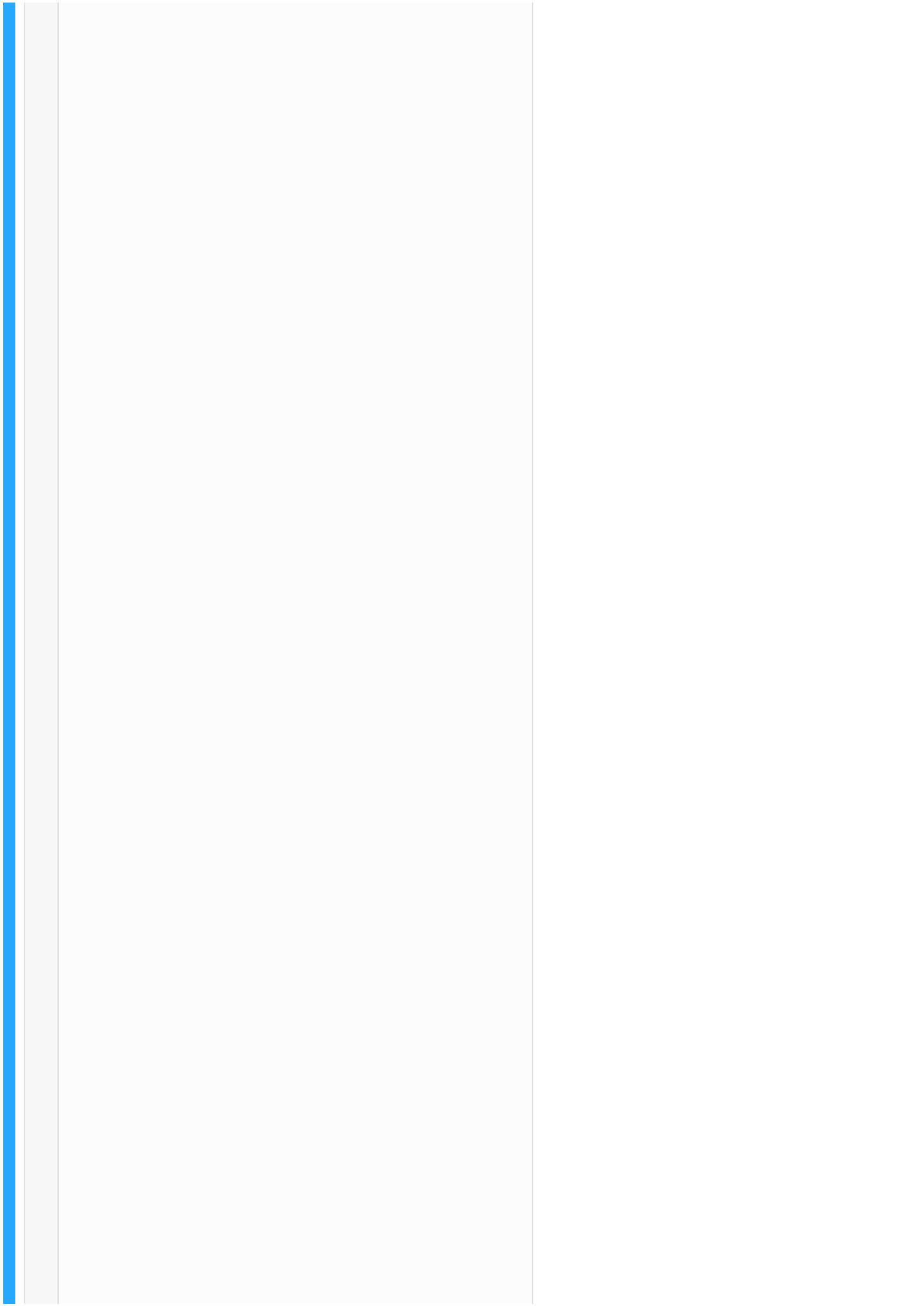


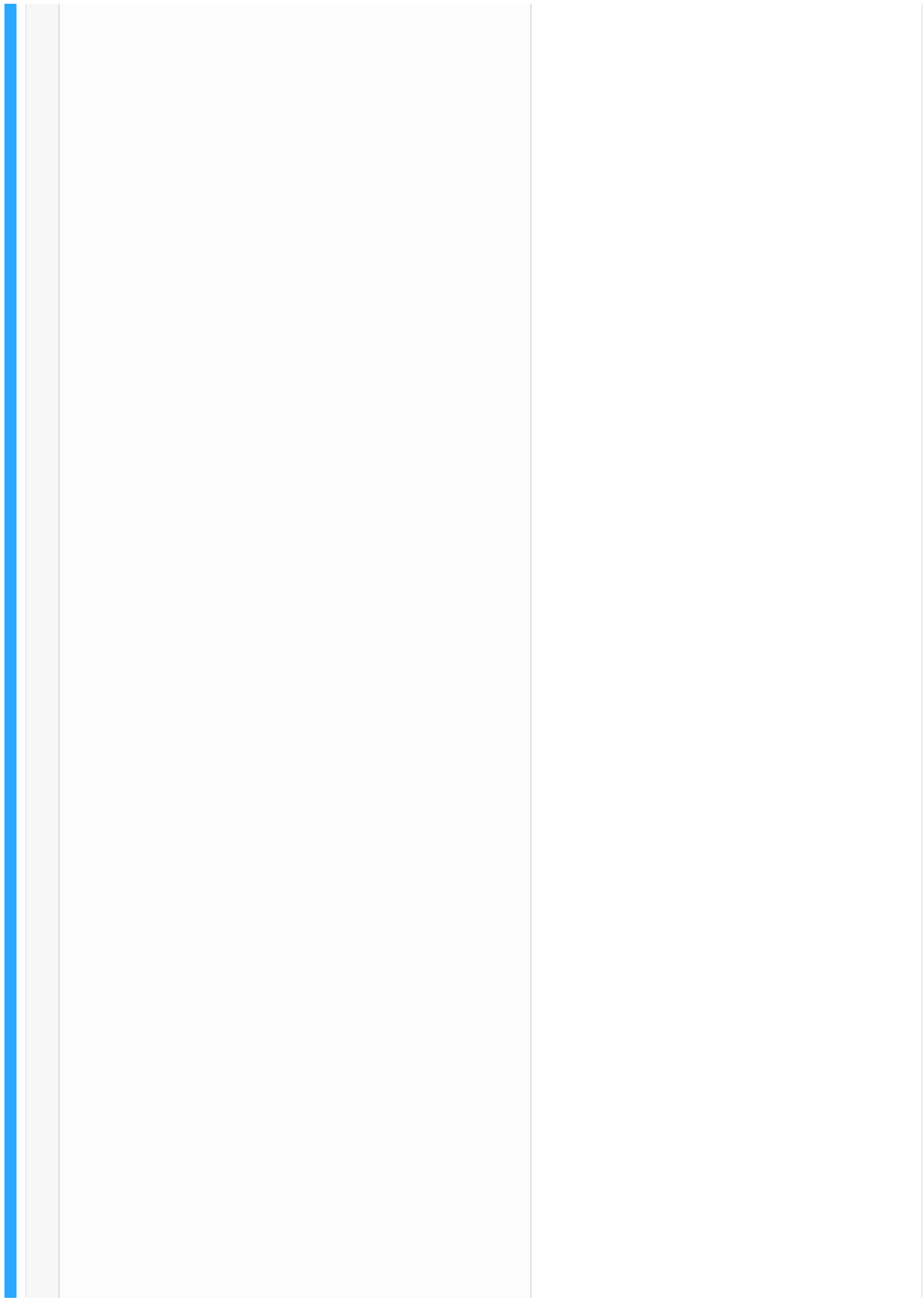




















DZone

Web Dev Zone

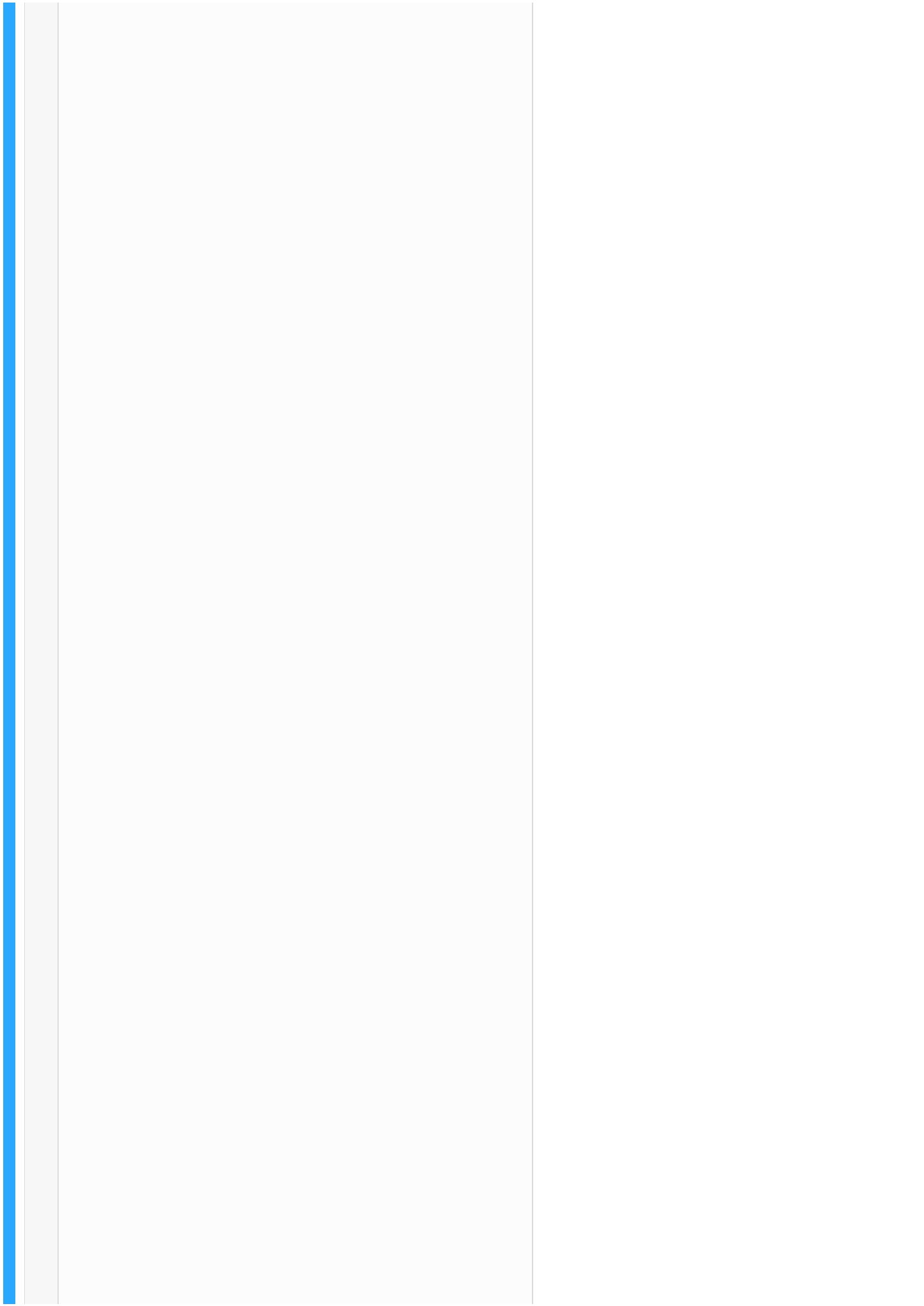
[Log In](#) / [Sign Up](#)

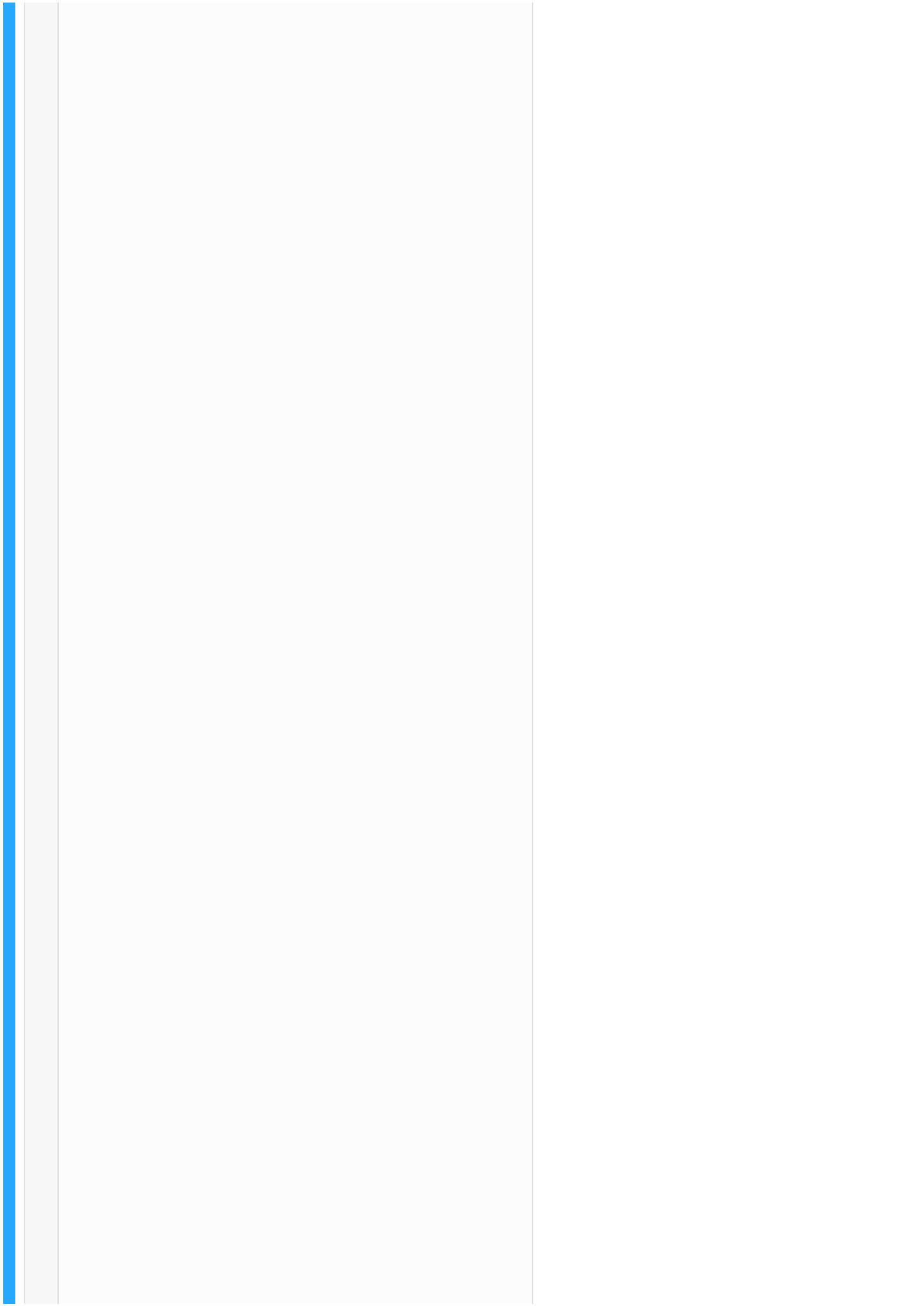


REFCARDZ GUIDES ZONES | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)

The `$request` is a little more complex as it's constructor signature looks like this:

```
1 public function __construct(  
2     $method,  
3     UriInterface $uri,  
4     HeadersInterface $headers,  
5     array $cookies,  
6     array $serverParams,  
7     StreamInterface $body,  
8     array $uploadedFiles = []  
9 )
```

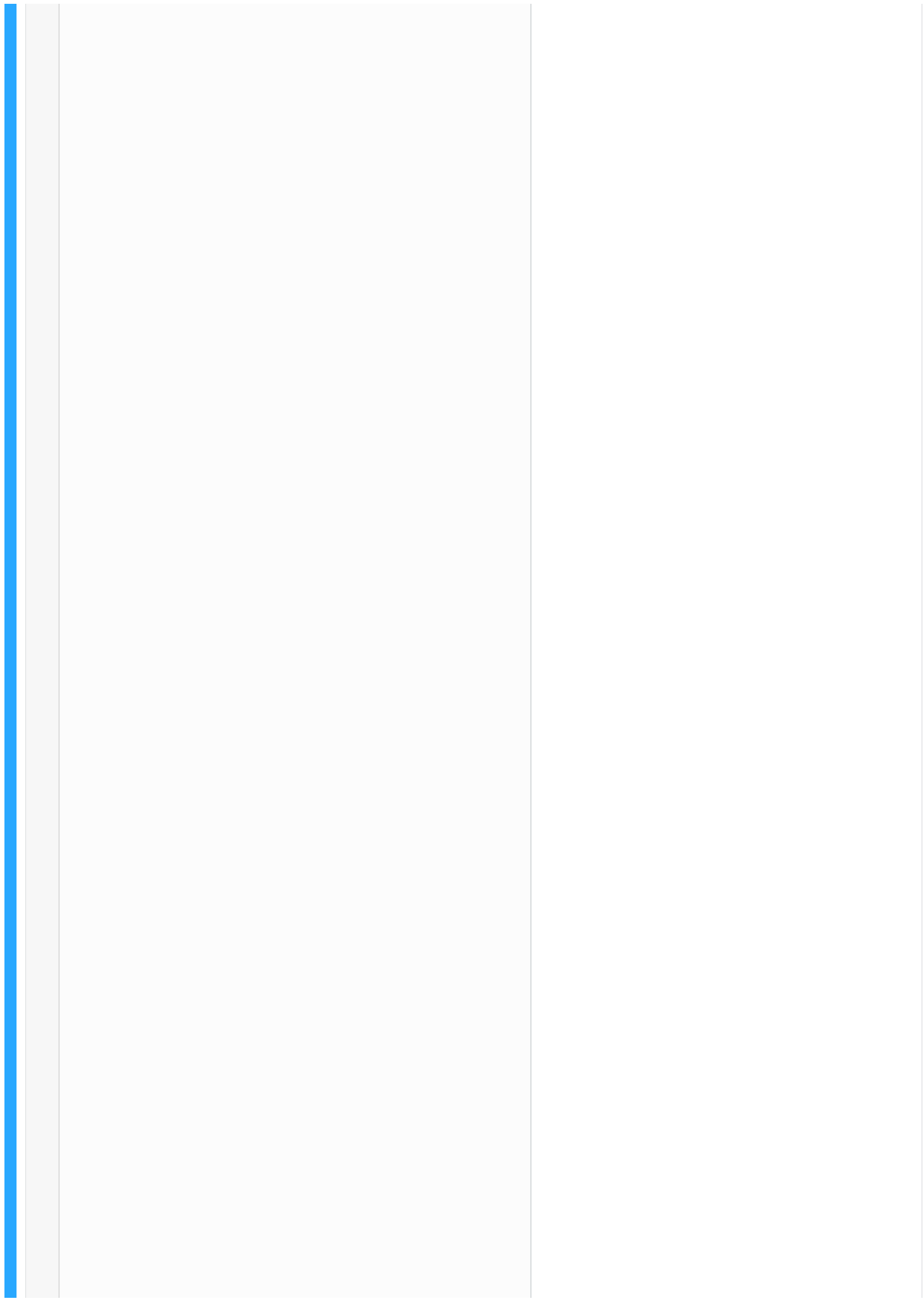










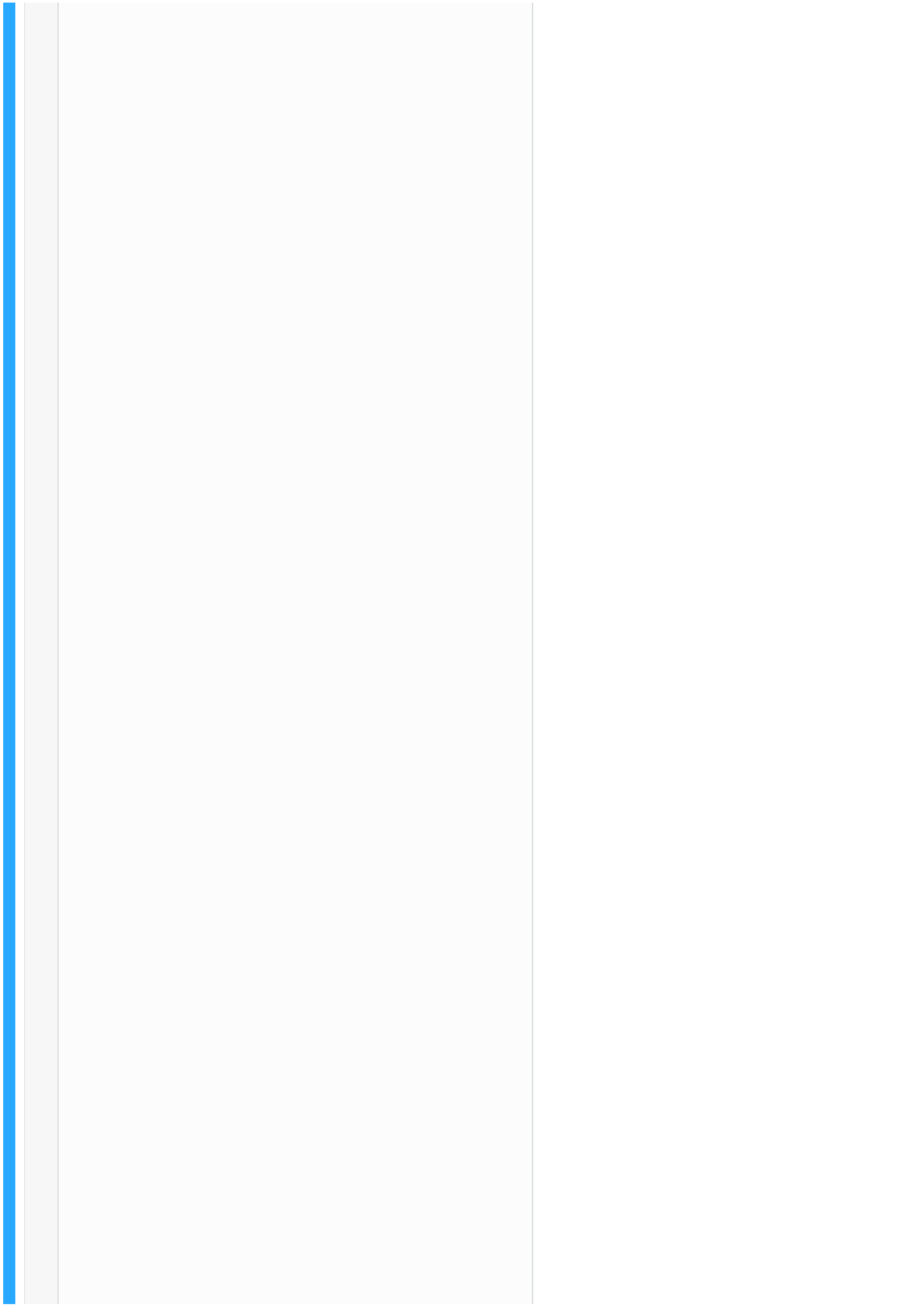




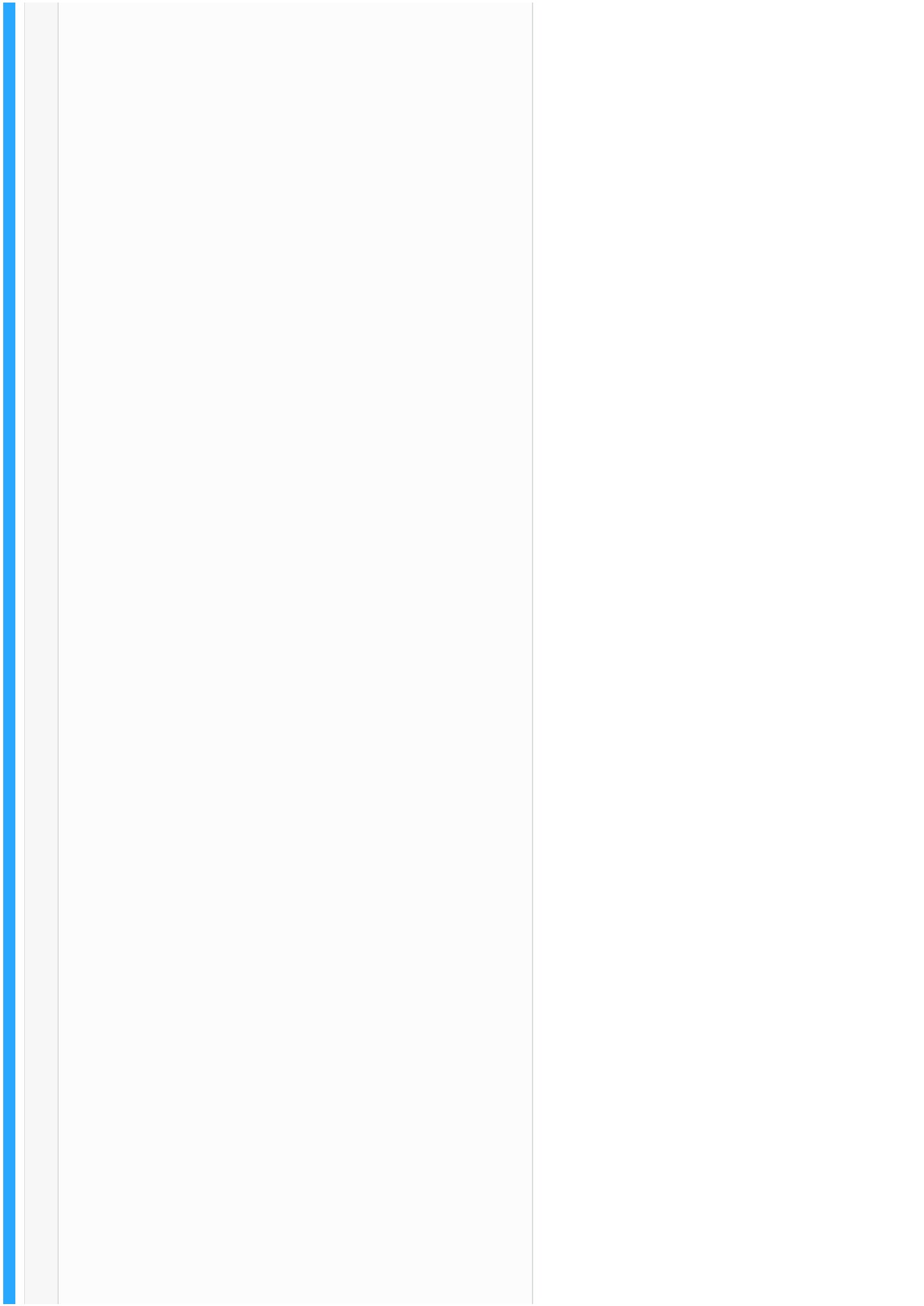












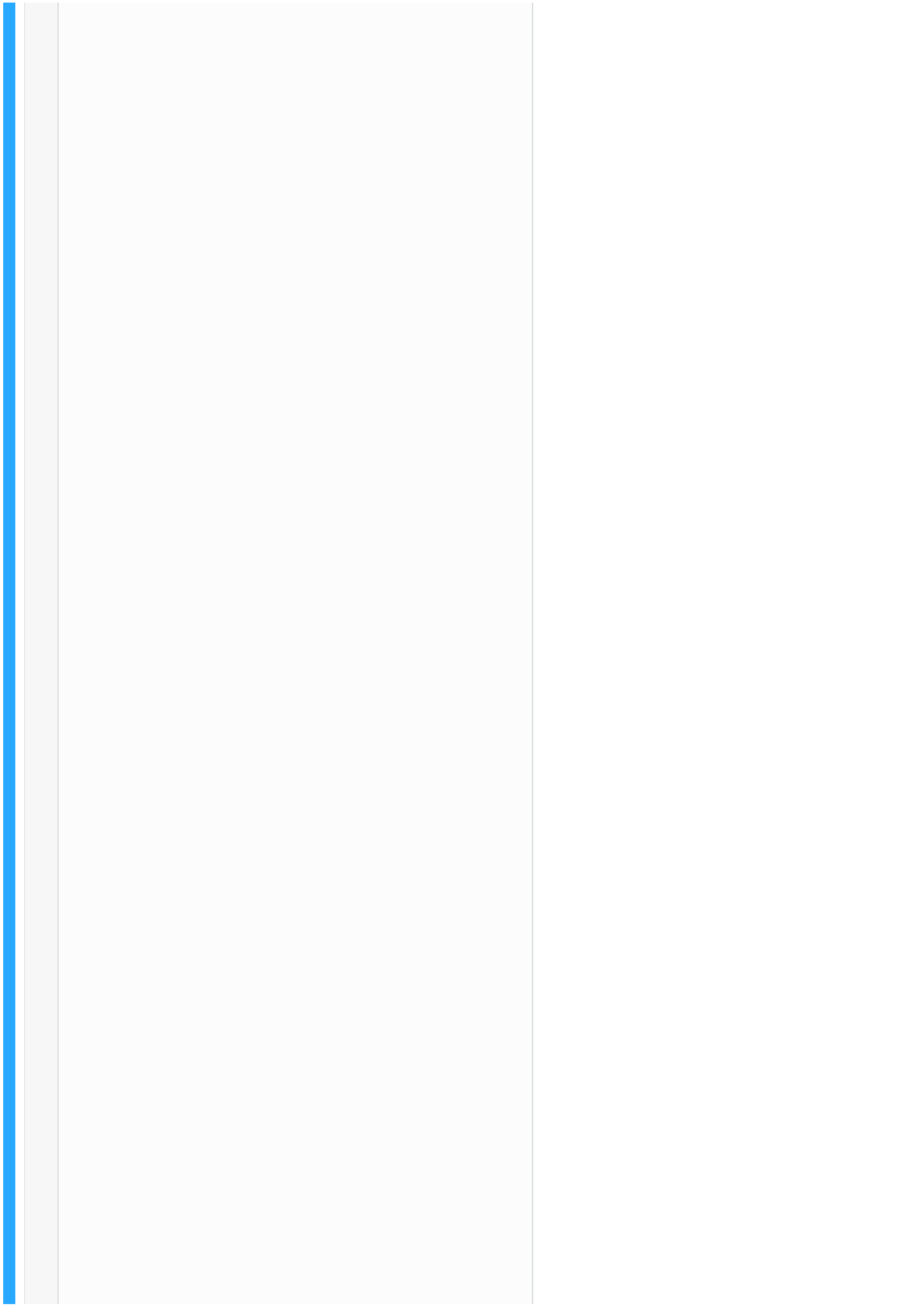


























DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)

However, Slim actually creates a `Slim\Http\Request` using the static `createFromEnvironment()` factory method, which takes a `Slim\Http\Environment` instance. Roughly, it does this:

```
1 $request = Request::createFromEnvironment(new Environment($_SERVER));
```





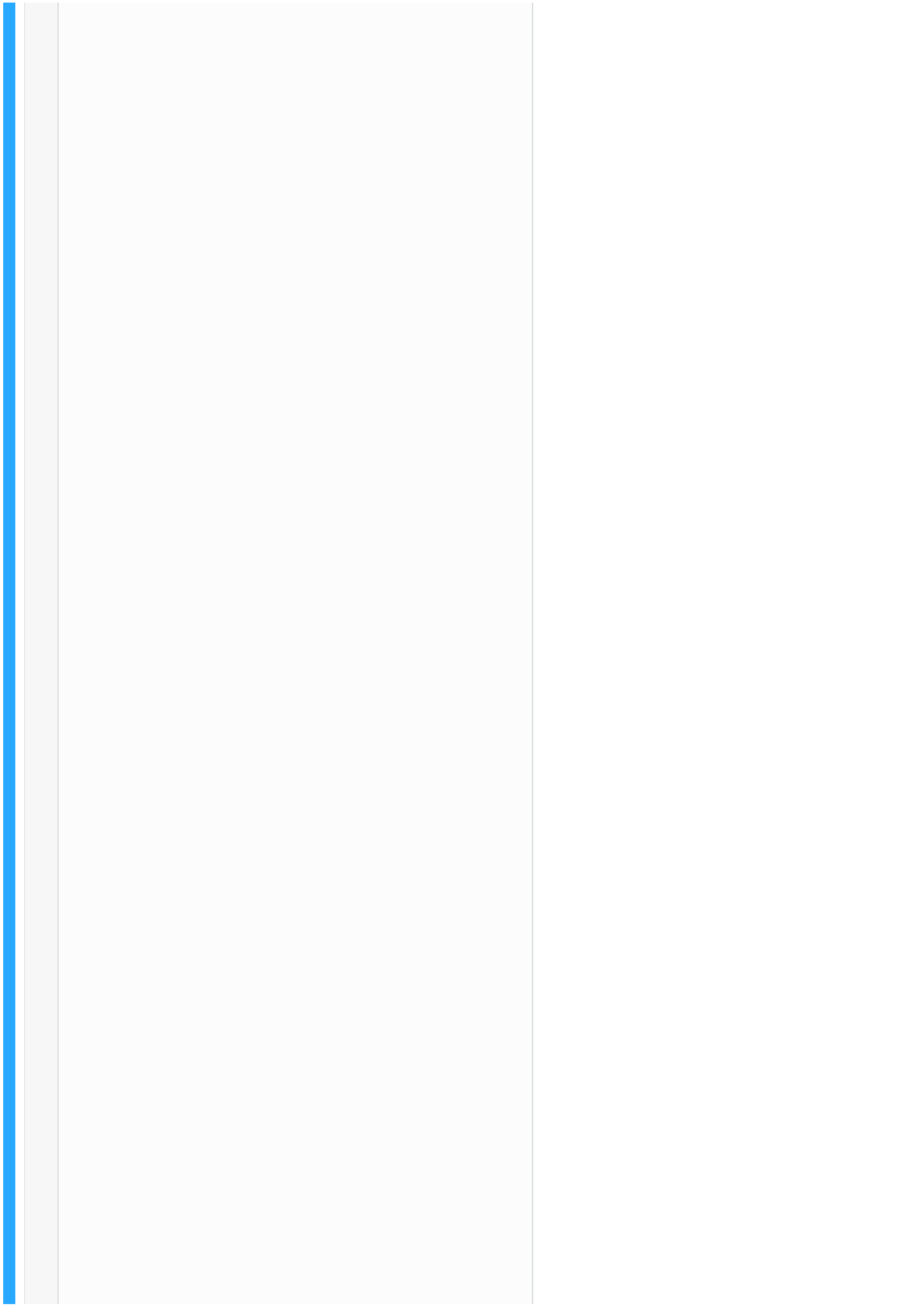




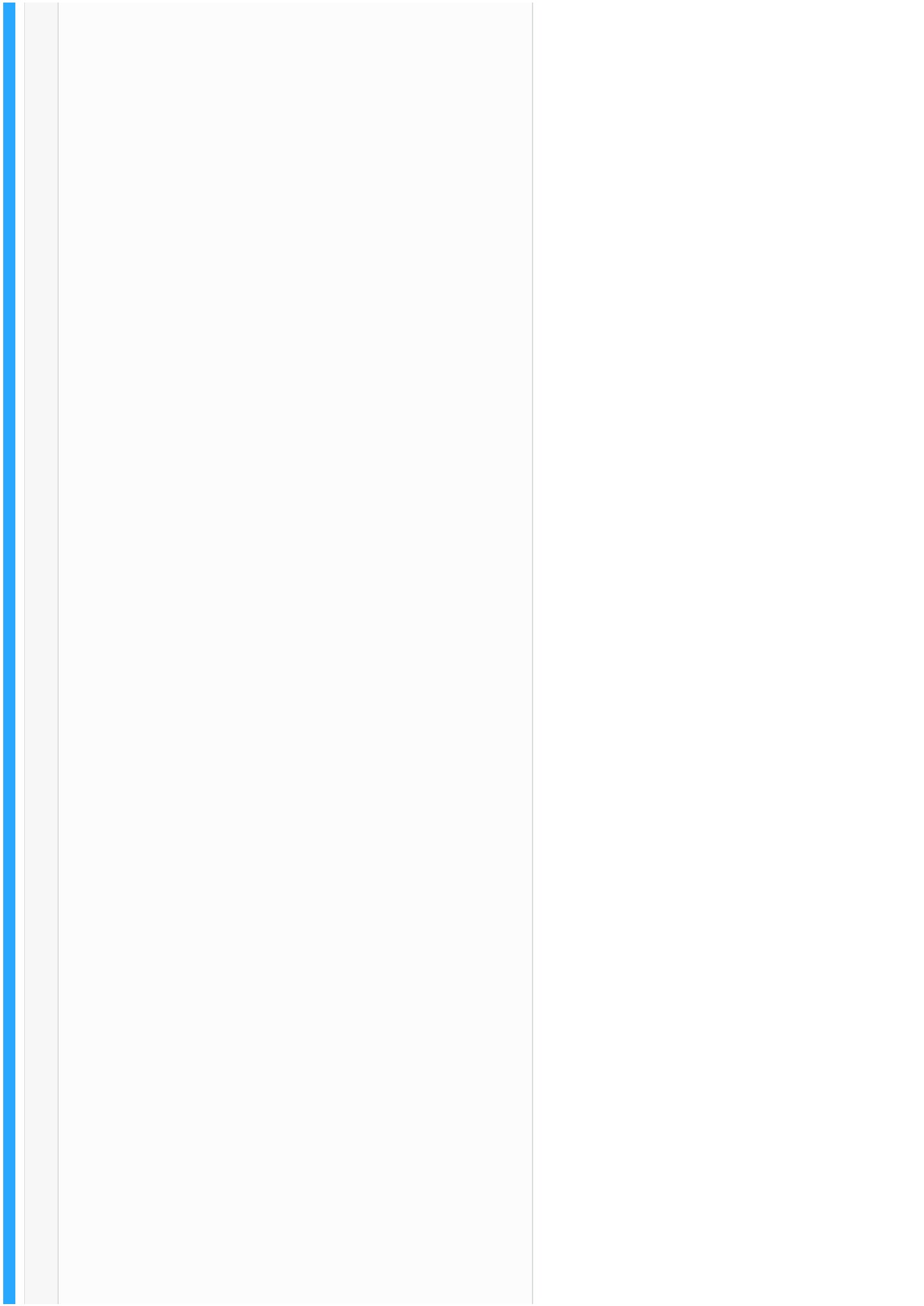














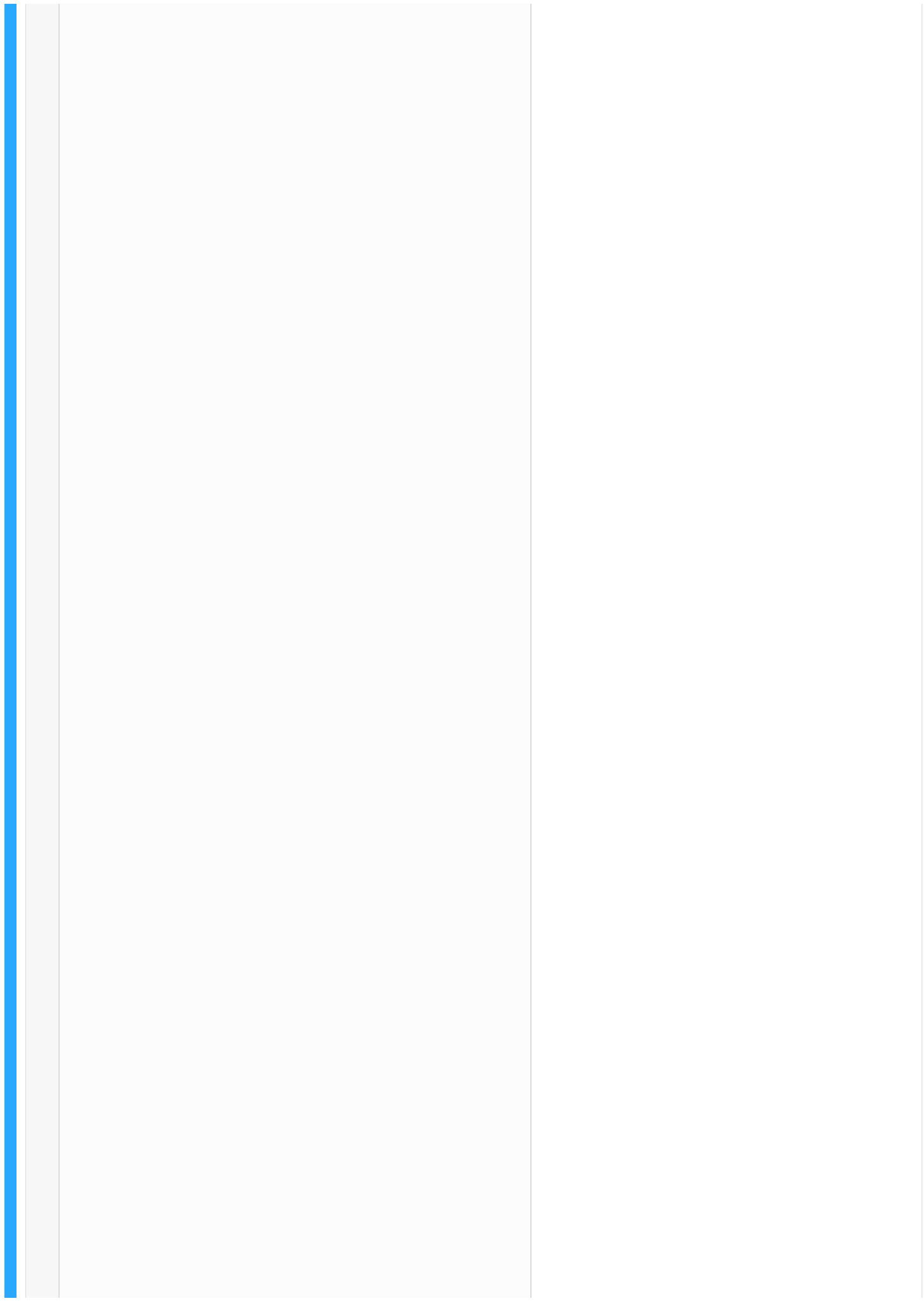








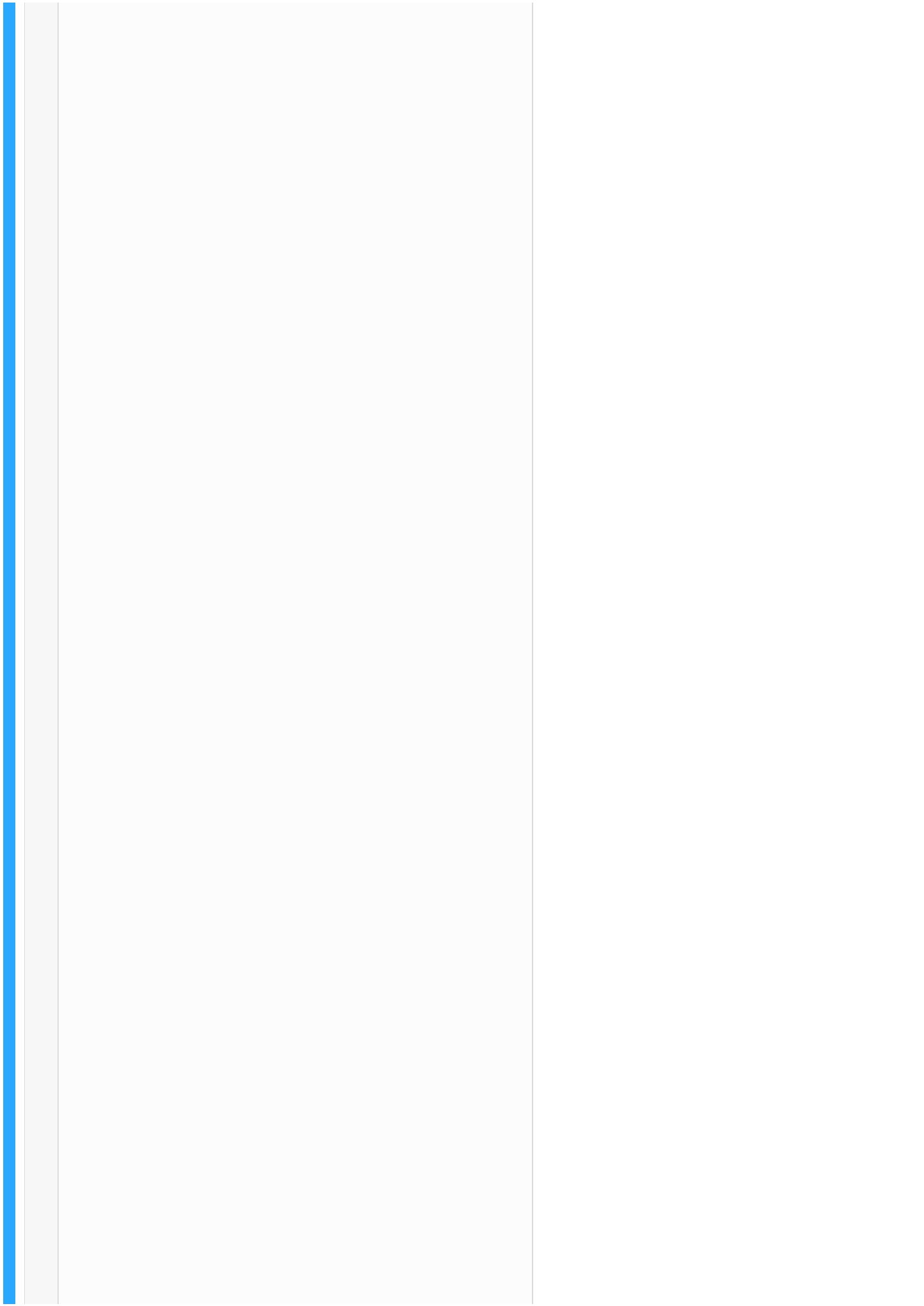


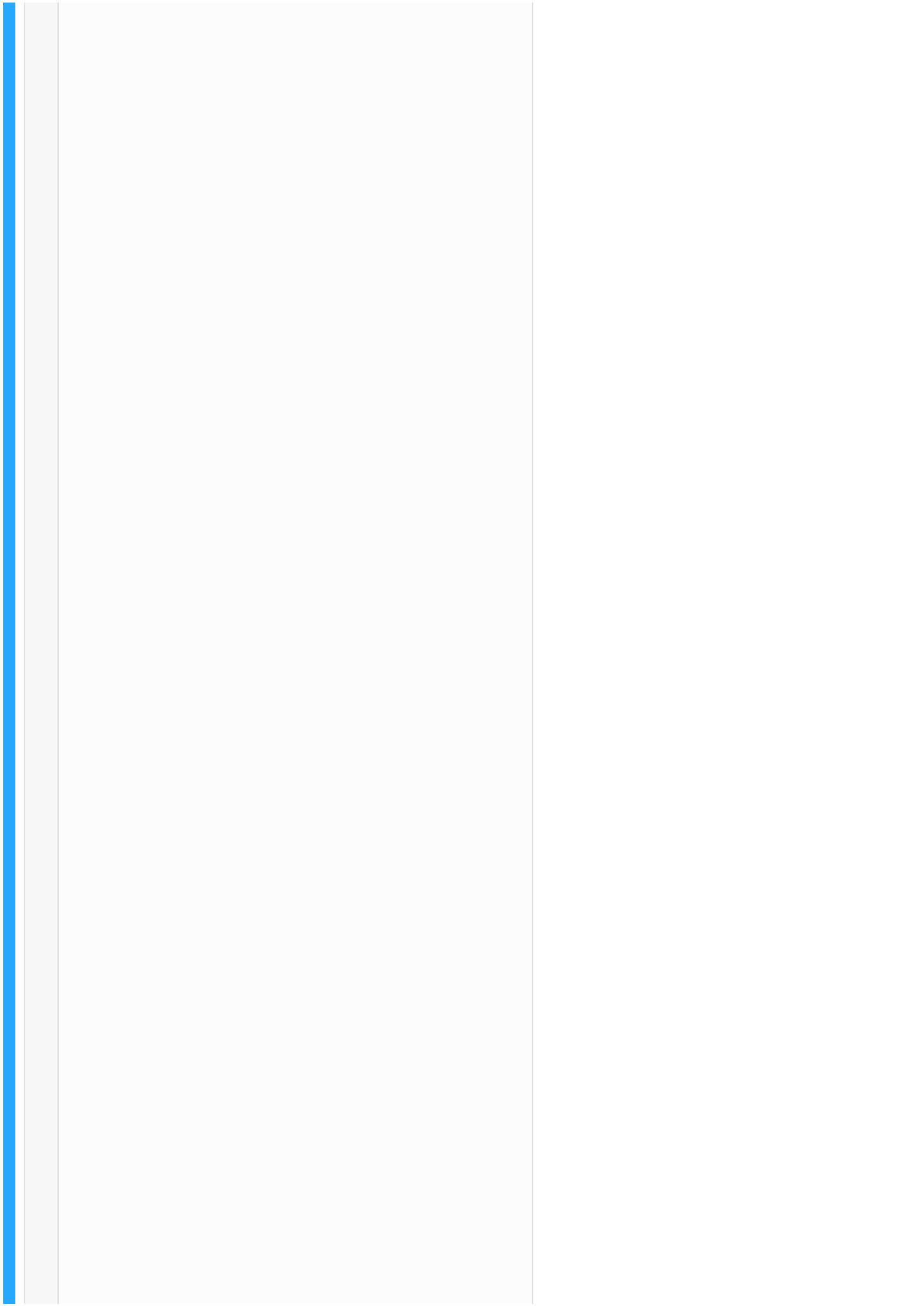






















DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)



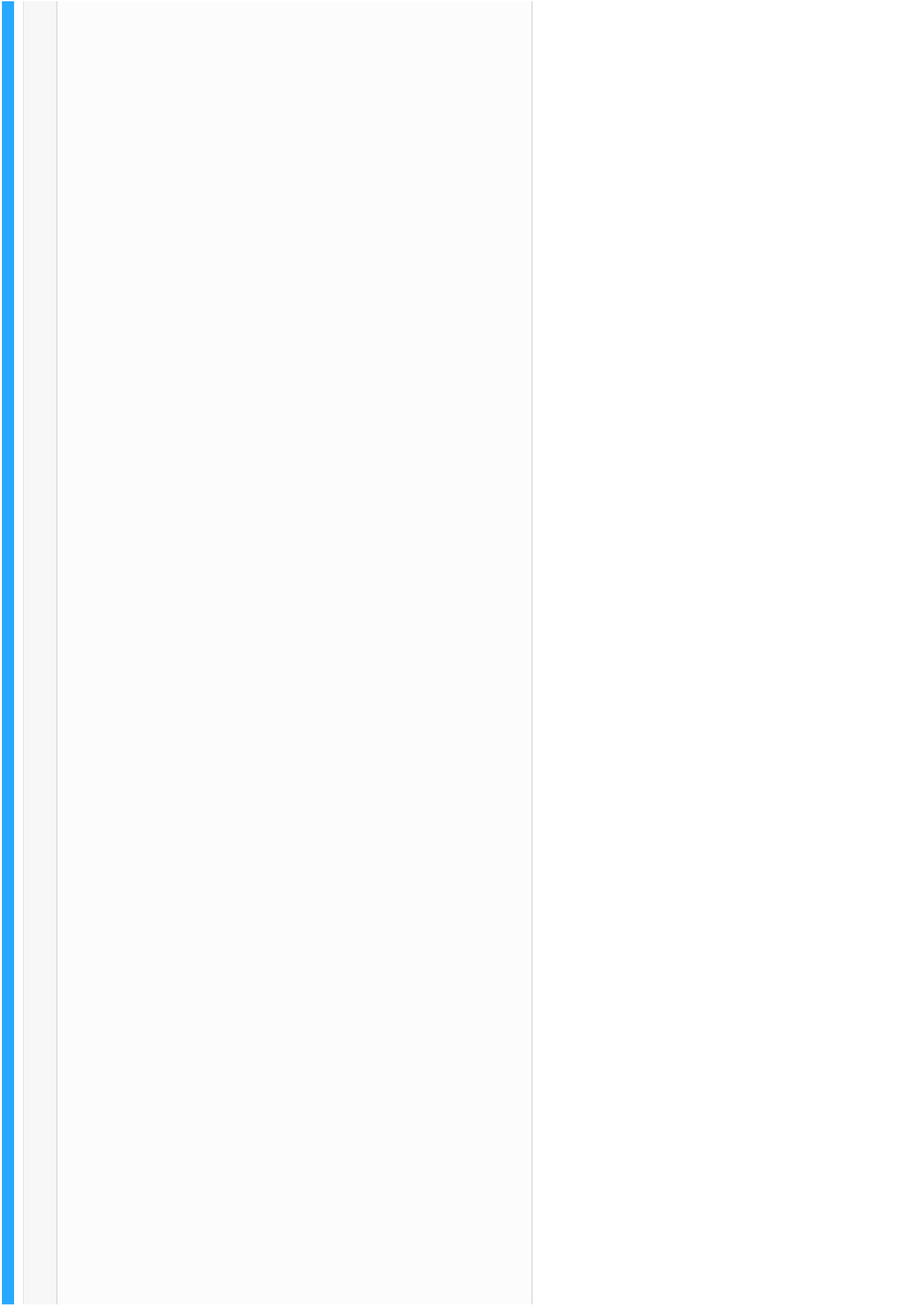
Setting up a `$_SERVER` array with the relevant elements can be a little tiresome in testing though. Fortunately, [Josh](#) needed to test Slim itself, so the Environment object has a handy static method called `mock()` that does this for us.

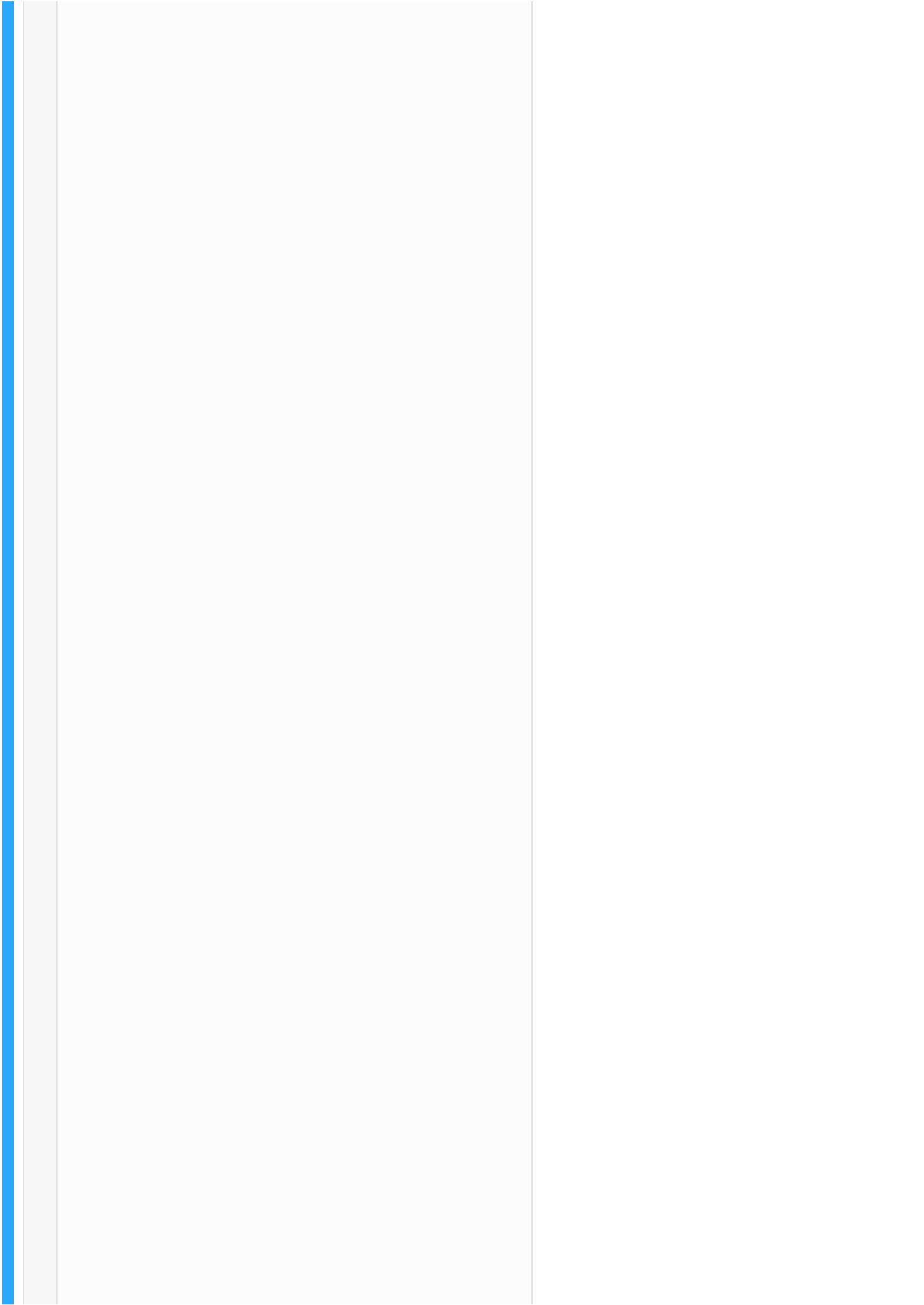
We use it like this:

```
1 $environment =  
  \Slim\Http\Environment::mock([  
2 'REQUEST_METHOD' => 'GET',  
3 'REQUEST_URI' => '/echo',  
4 'QUERY_STRING' => 'foo=bar'  
5 ]);
```



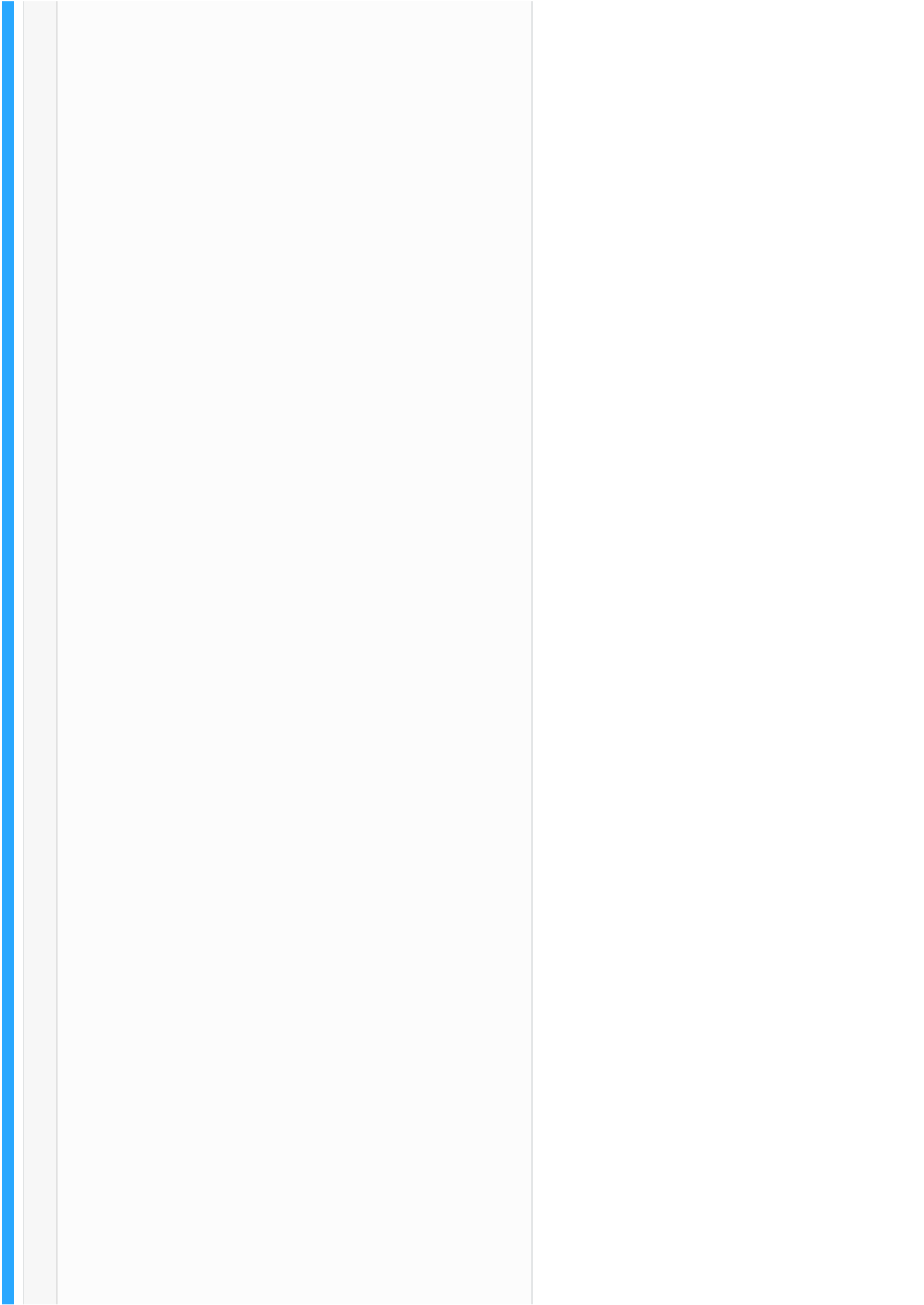


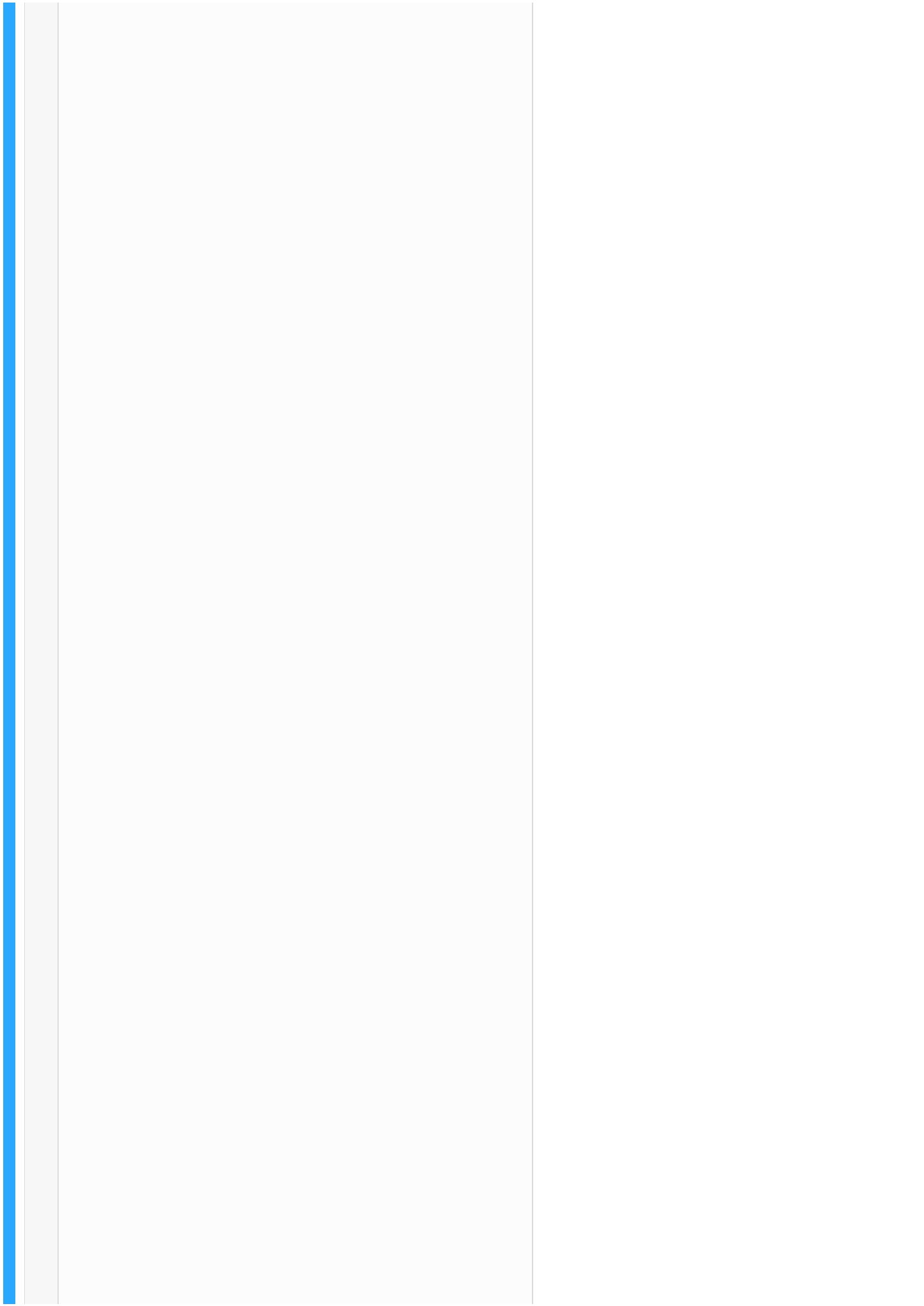


























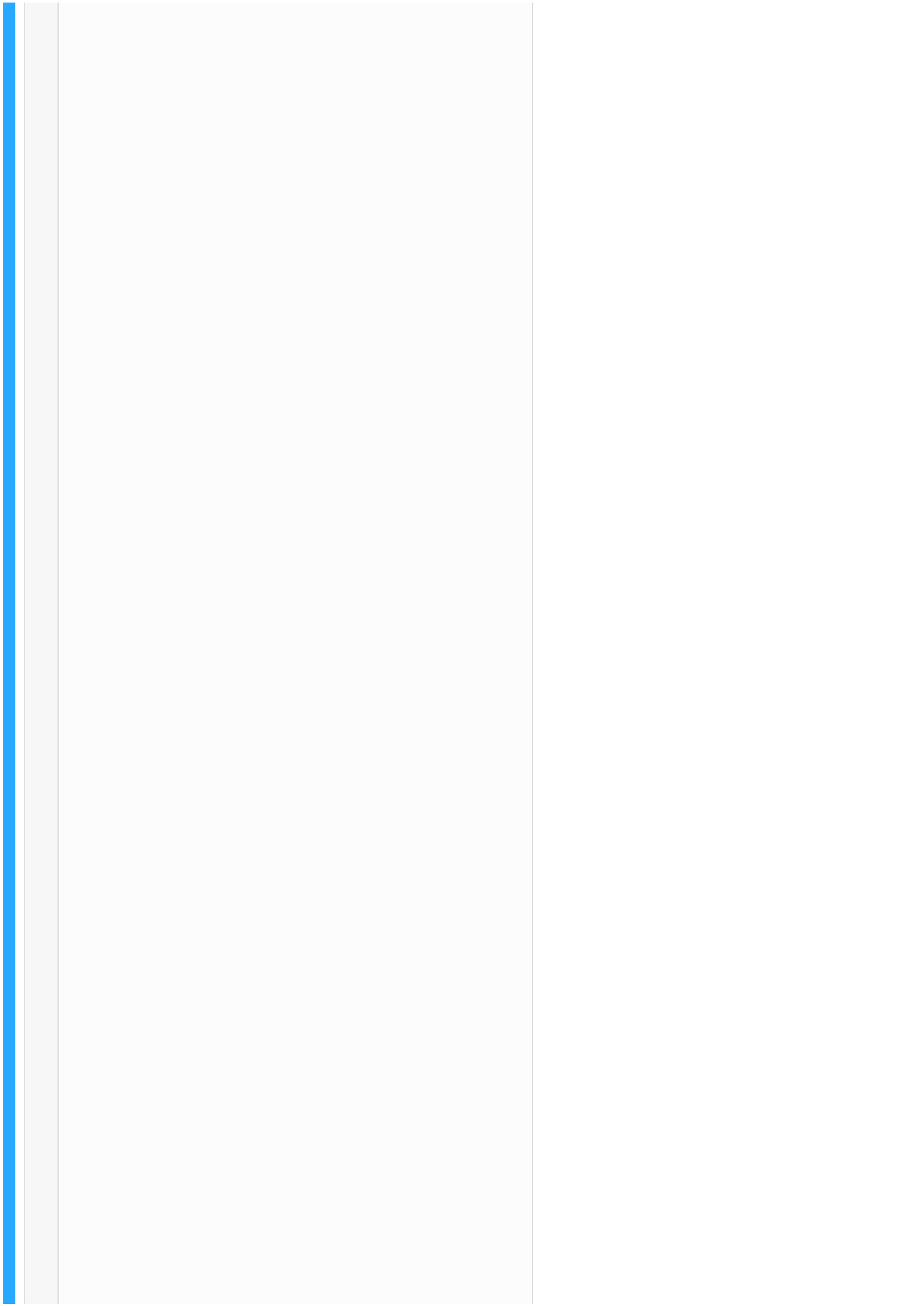


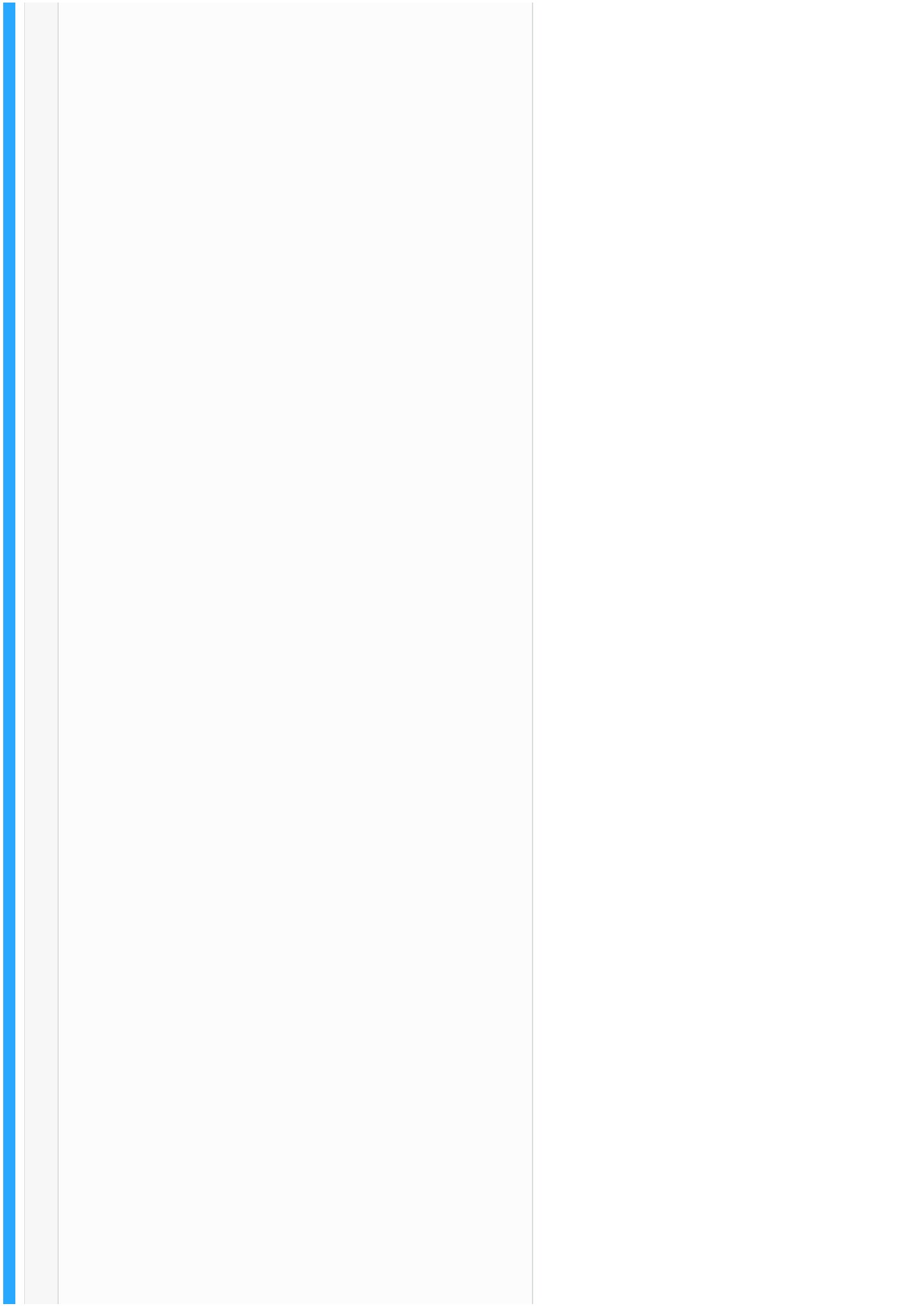




















DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#)

[GUIDES](#)

[ZONES](#)

[Agile](#)

[AI](#)

[Big Data](#)

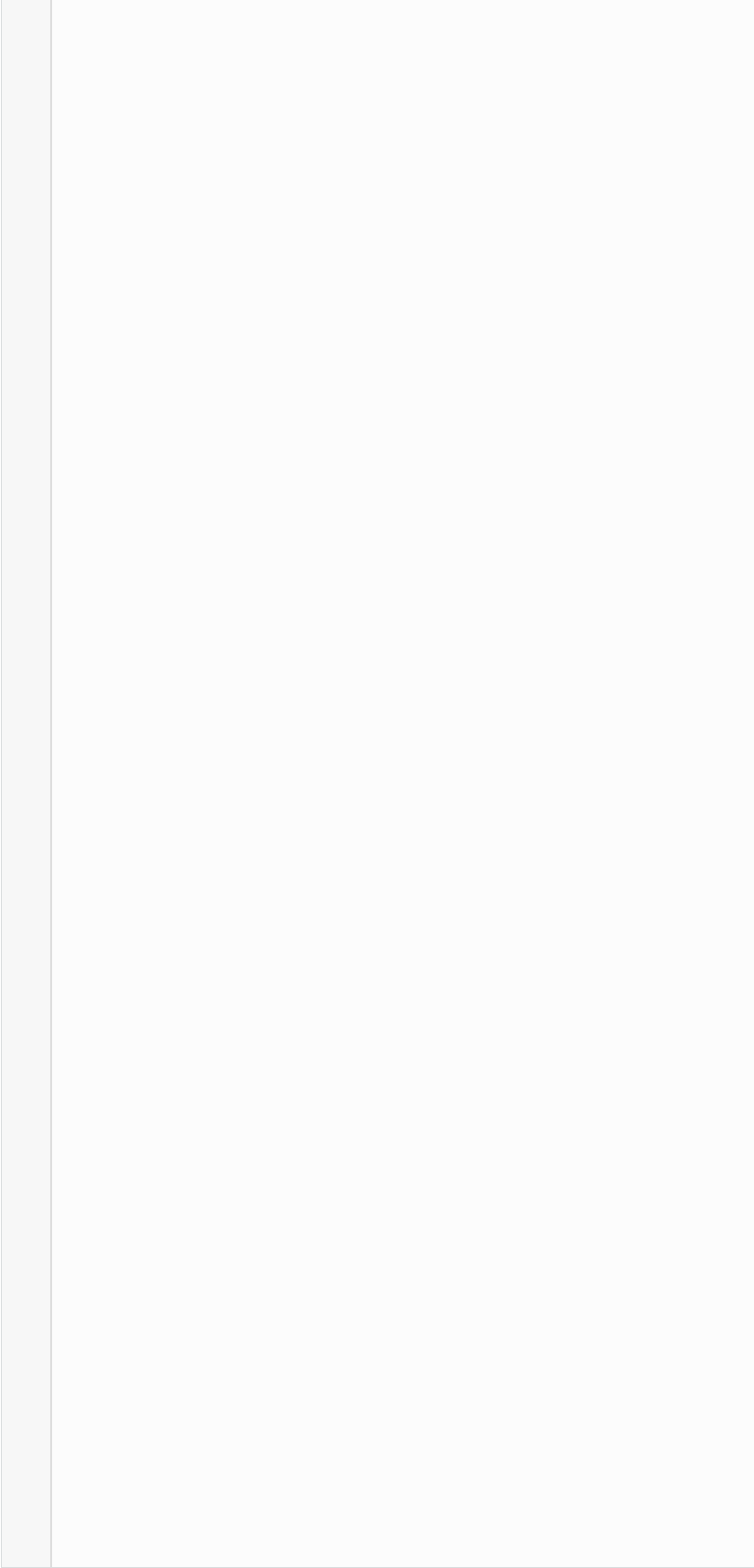
[Cloud](#)

[Database](#)

[DevOps](#)

[Integration](#)

[IoT](#)



As you can see, `mock()` takes an array which contains `$_SERVER` keys that we wish to set up for our particular test. Usually we set the `REQUEST_METHOD`, `REQUEST_URI` and, if we need it, `QUERY_STRING`. We need `QUERY_STRING` as this is the key in `$_SERVER` that Request uses to determine the query parameters for the request.

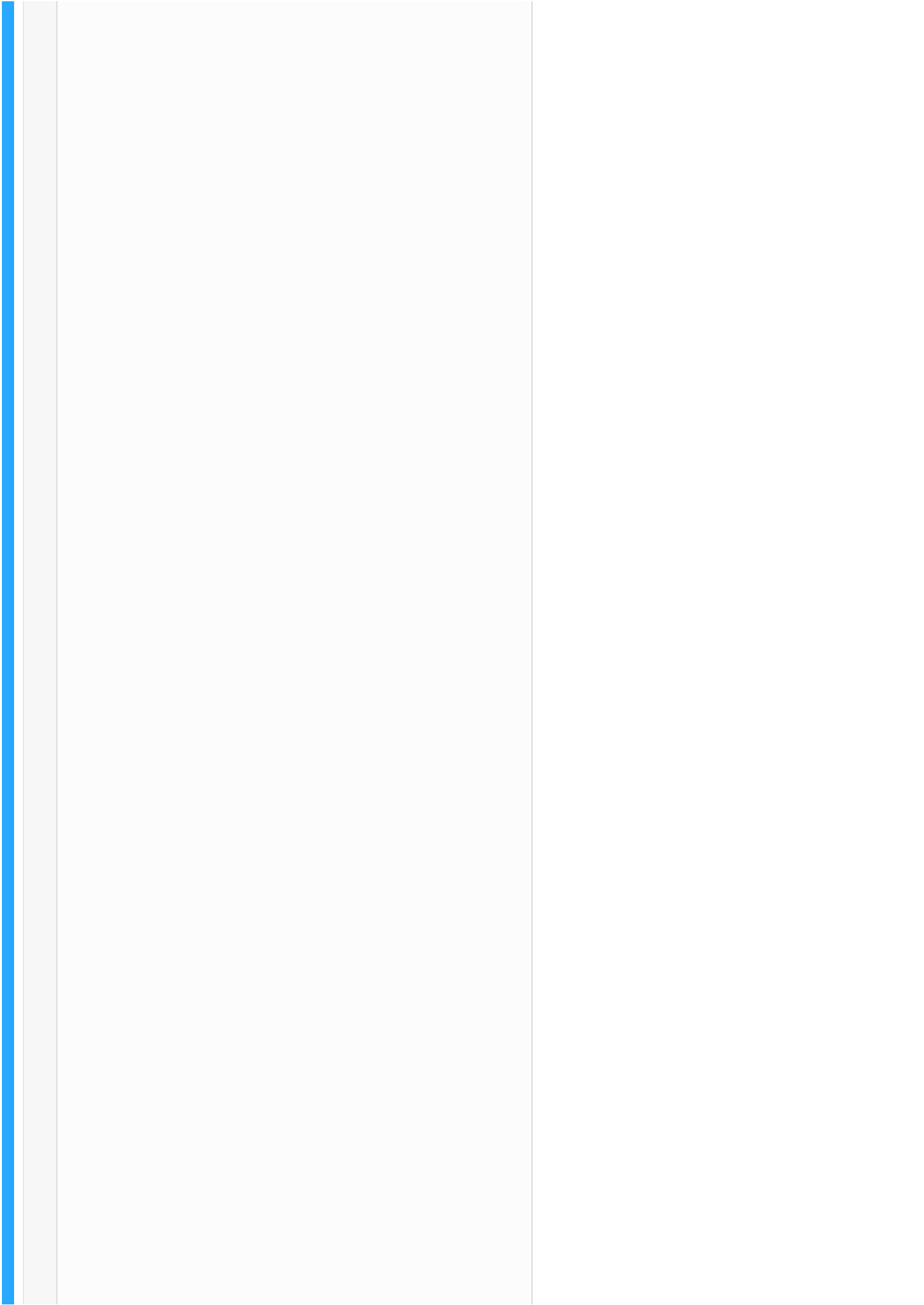
Putting it All Together

Hence, our completed test looks like this:

```
1 class EchoActionTest extends \PHPUnit_Framework_
   TestCase
2 {
3     public function testGetRequestReturnsEcho()
4     {
5         // instantiate action
6         $action = new \App\Action\EchoAction();
7
8         // We need a request and response object to invo
           ke the action
9         $environment =
           \Slim\Http\Environment::mock([
10     'REQUEST_METHOD' => 'GET',
11     'REQUEST_URI' => '/echo',
12     'QUERY_STRING'=>'foo=bar']
13 );
14     $request = \Slim\Http\Request::createFromEnviron
           ment($environment);
15     $response = new \Slim\Http\Response();
16
17     // run the controller action and test it
18     $response = $action($request, $response,
           []);
19     $this->assertSame((string)$response->getBody(),
           '{"foo":"bar"}');
20 }
21 }
```





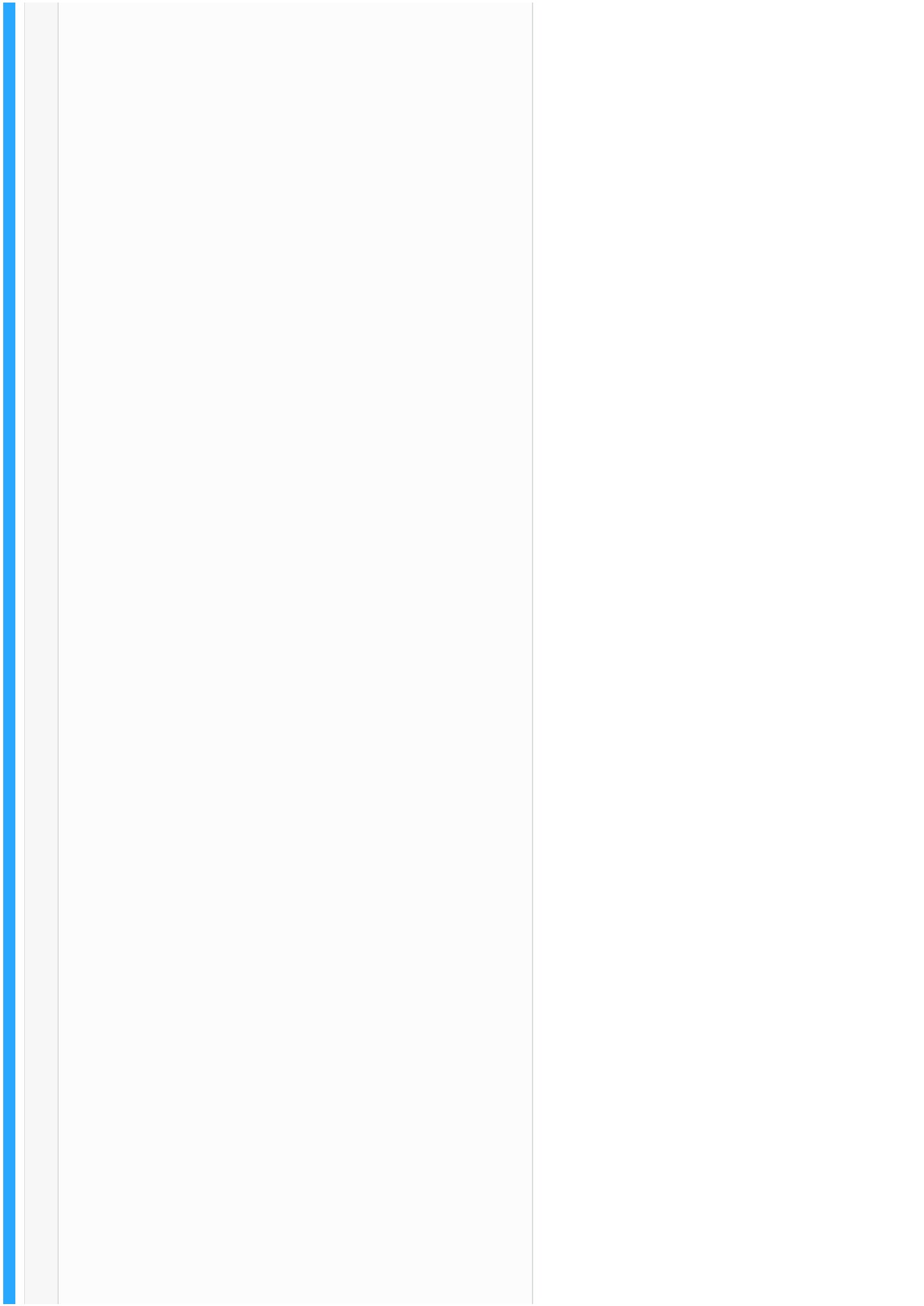


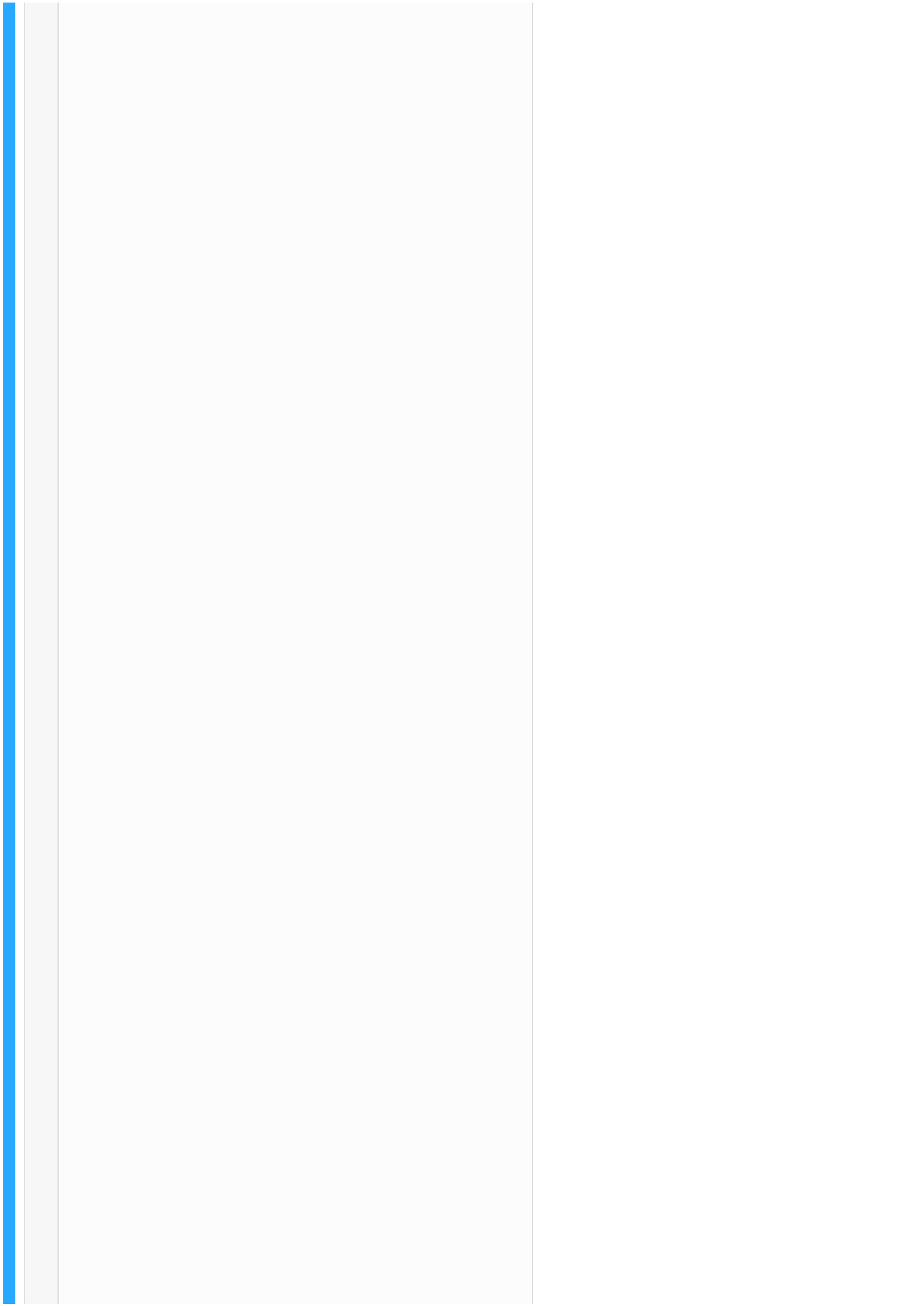








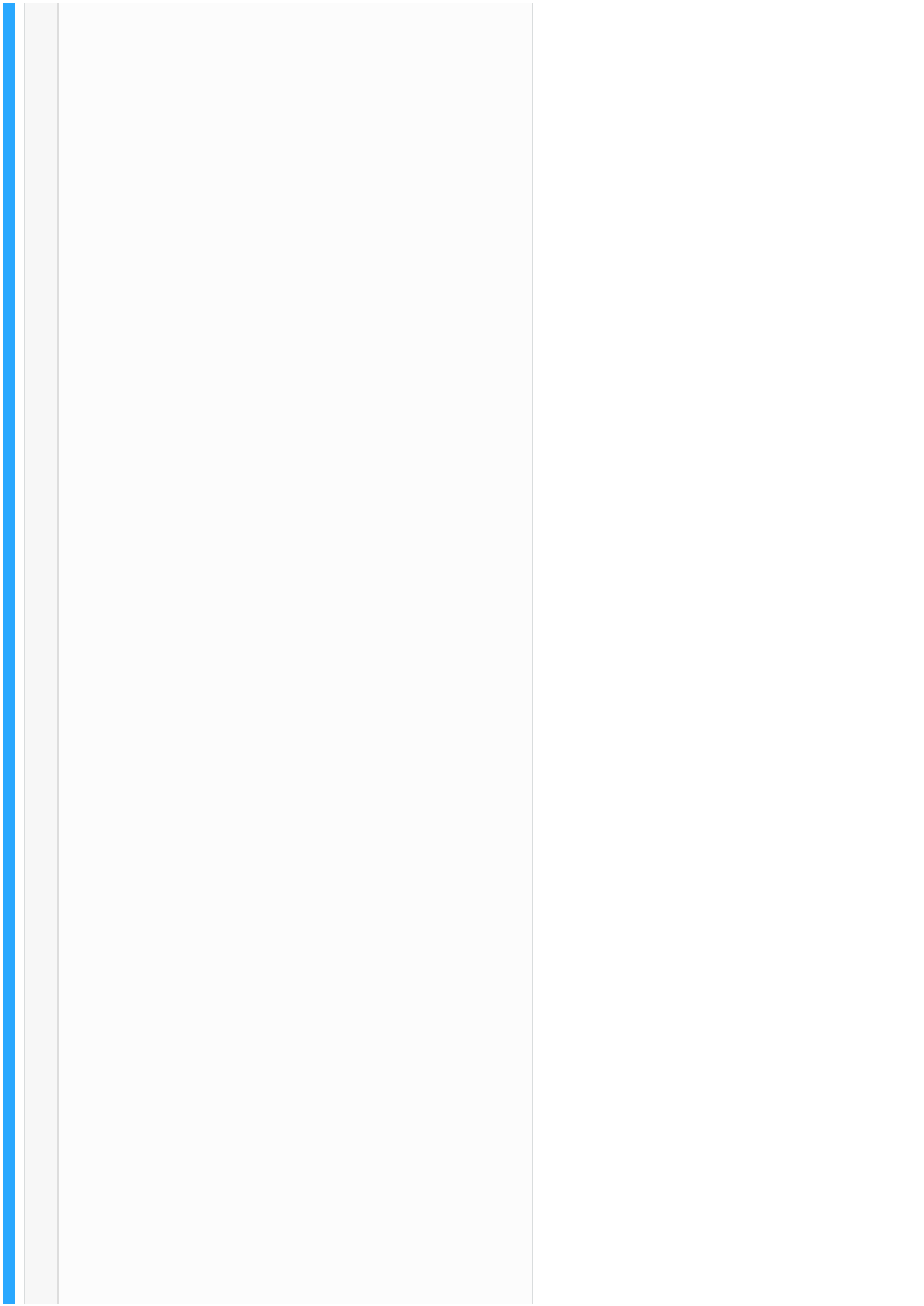




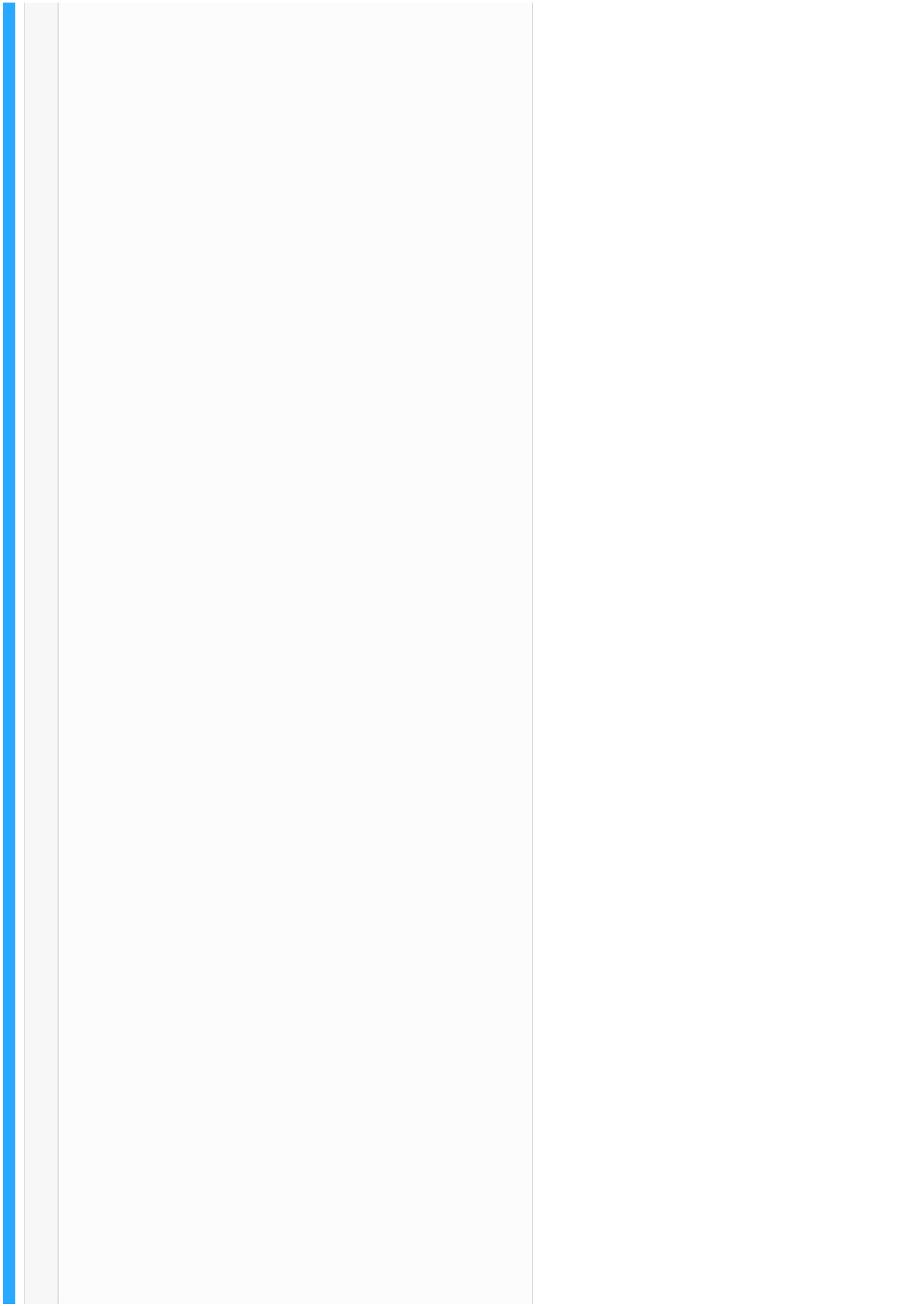




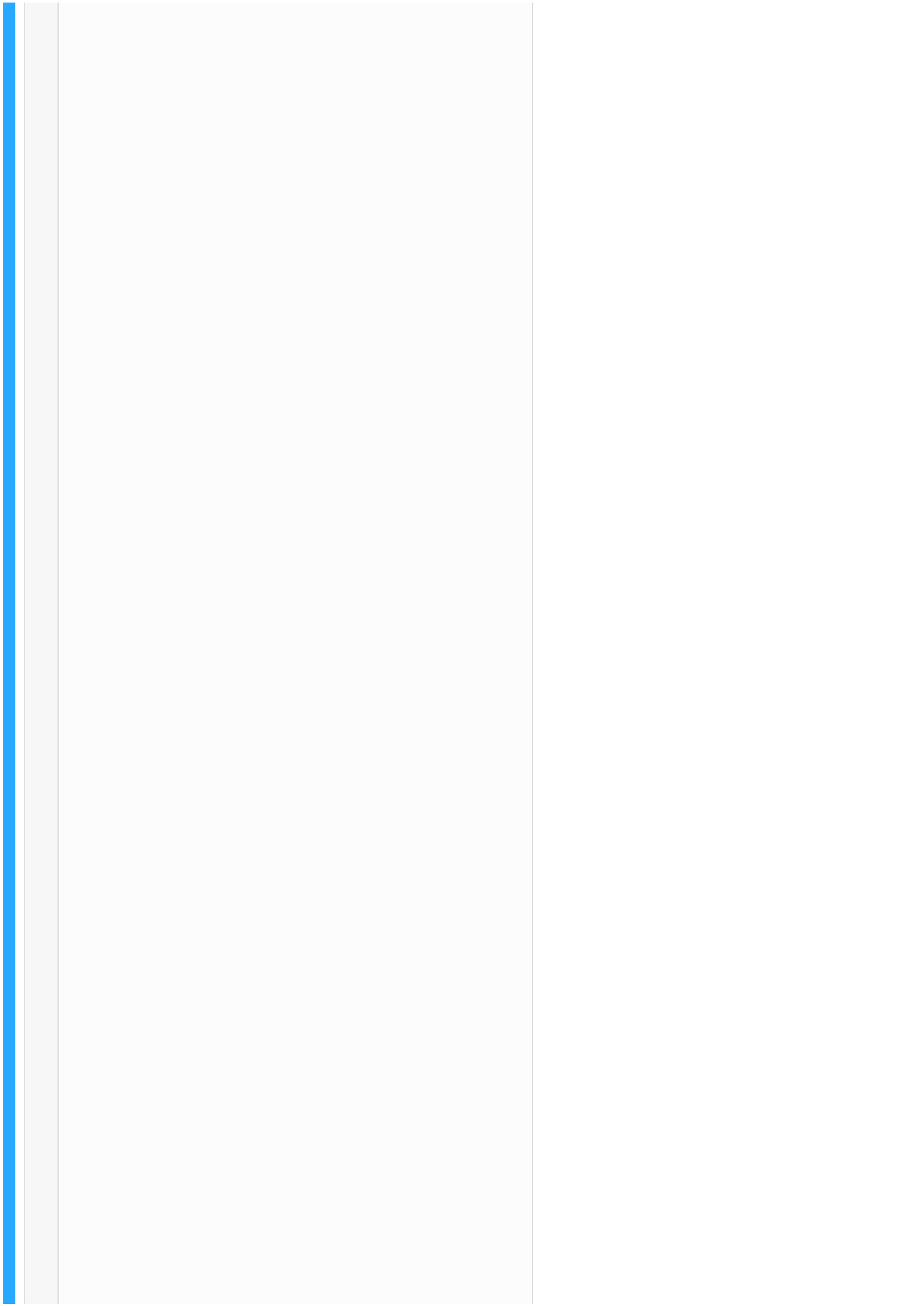




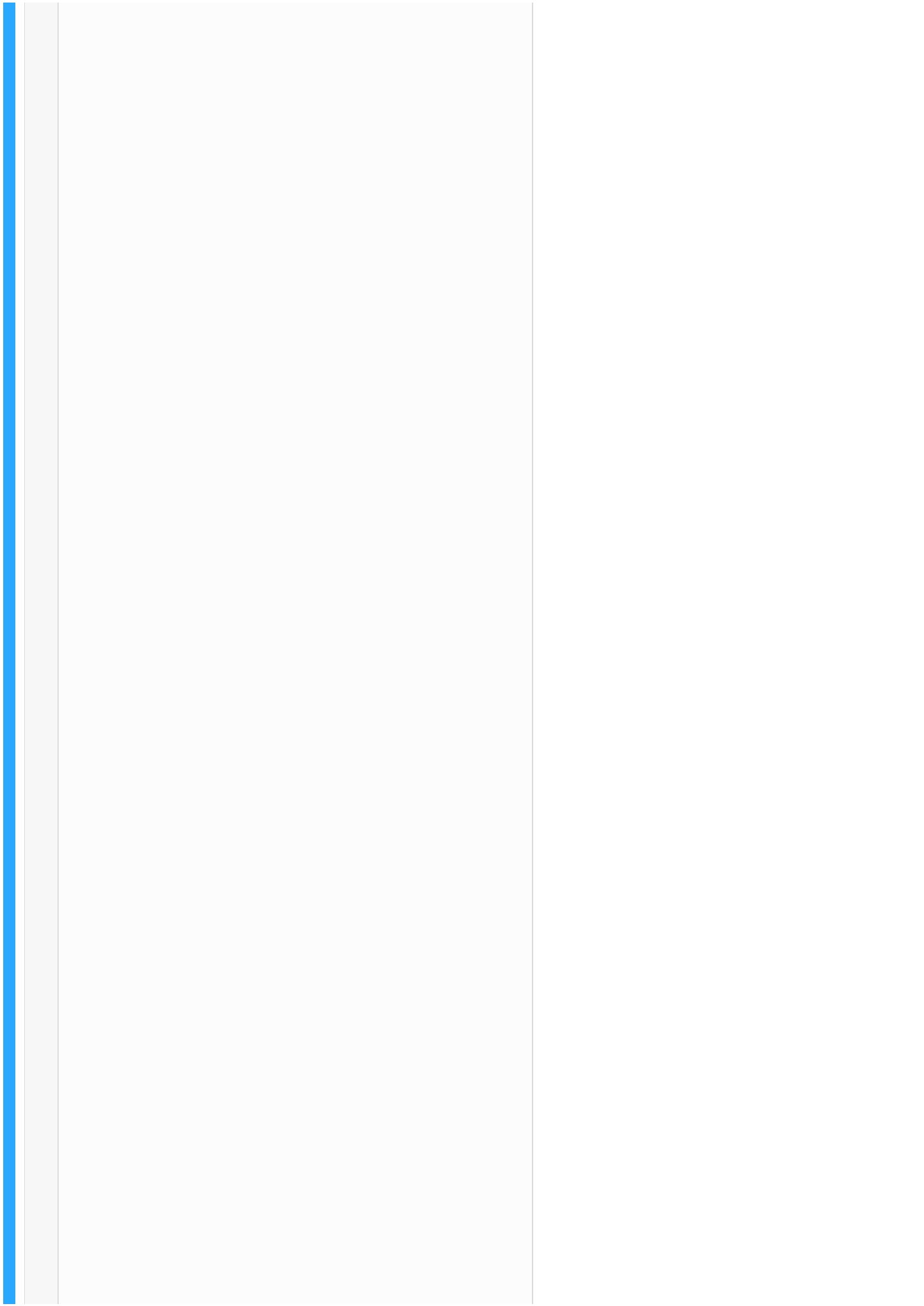


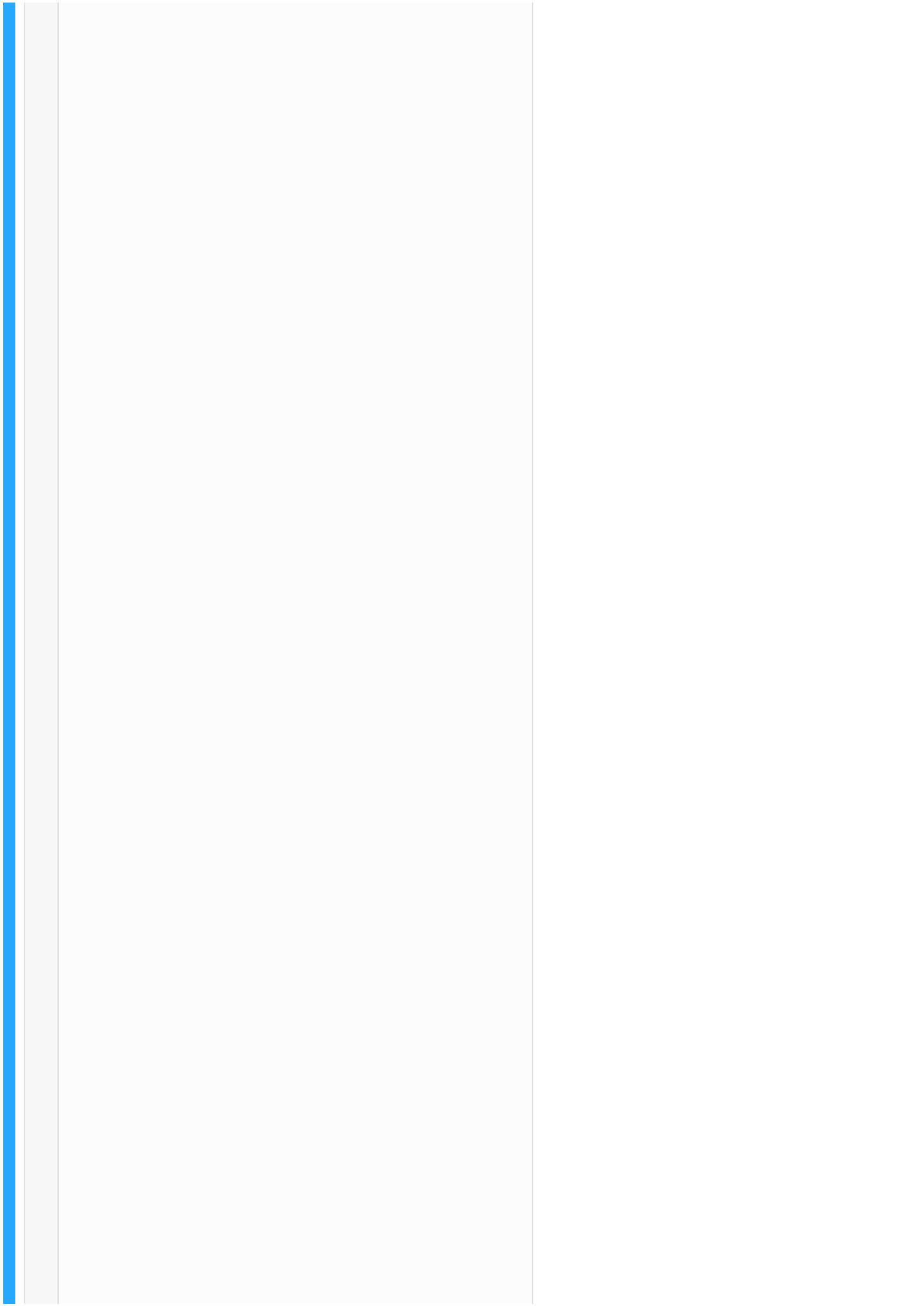


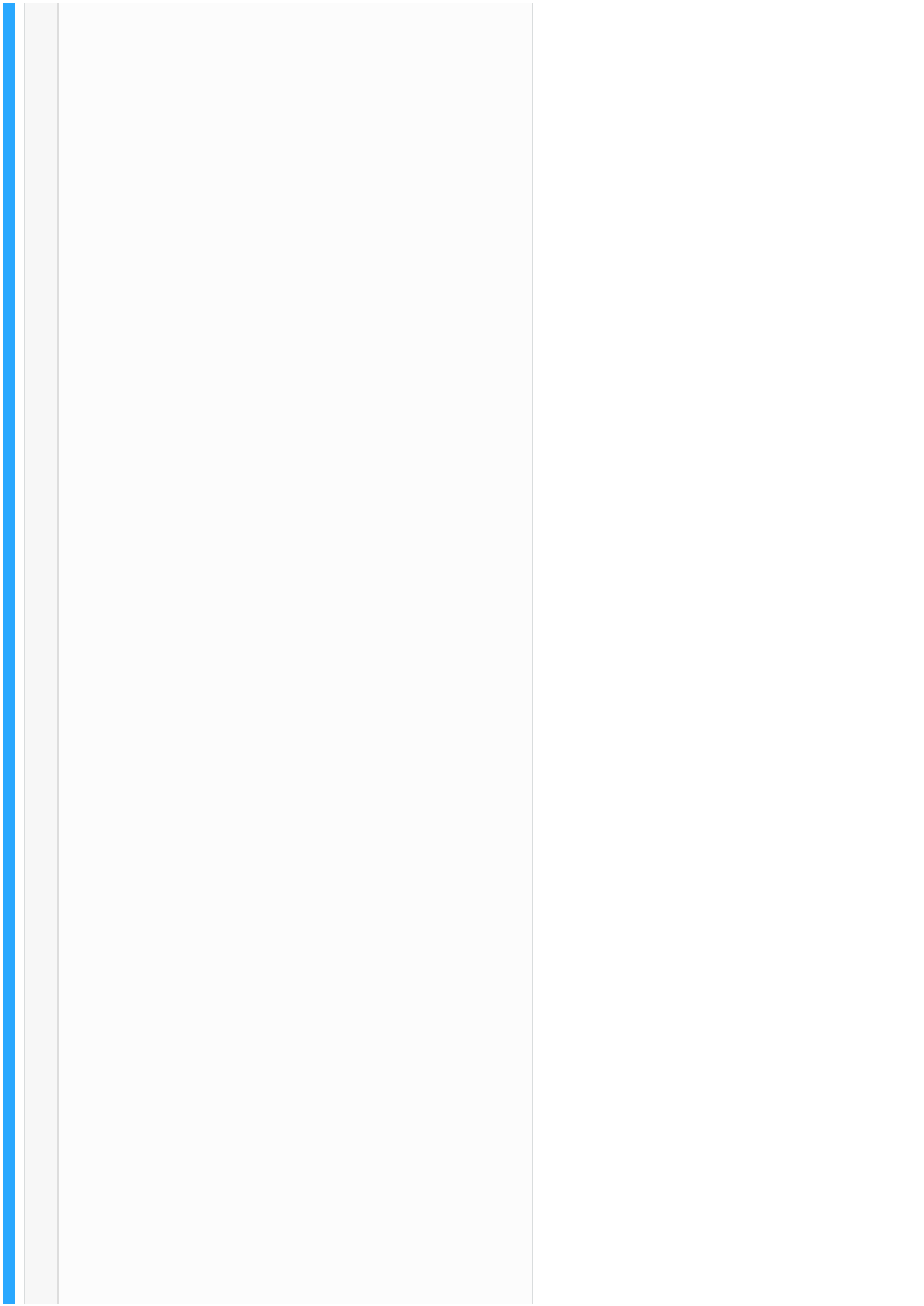






















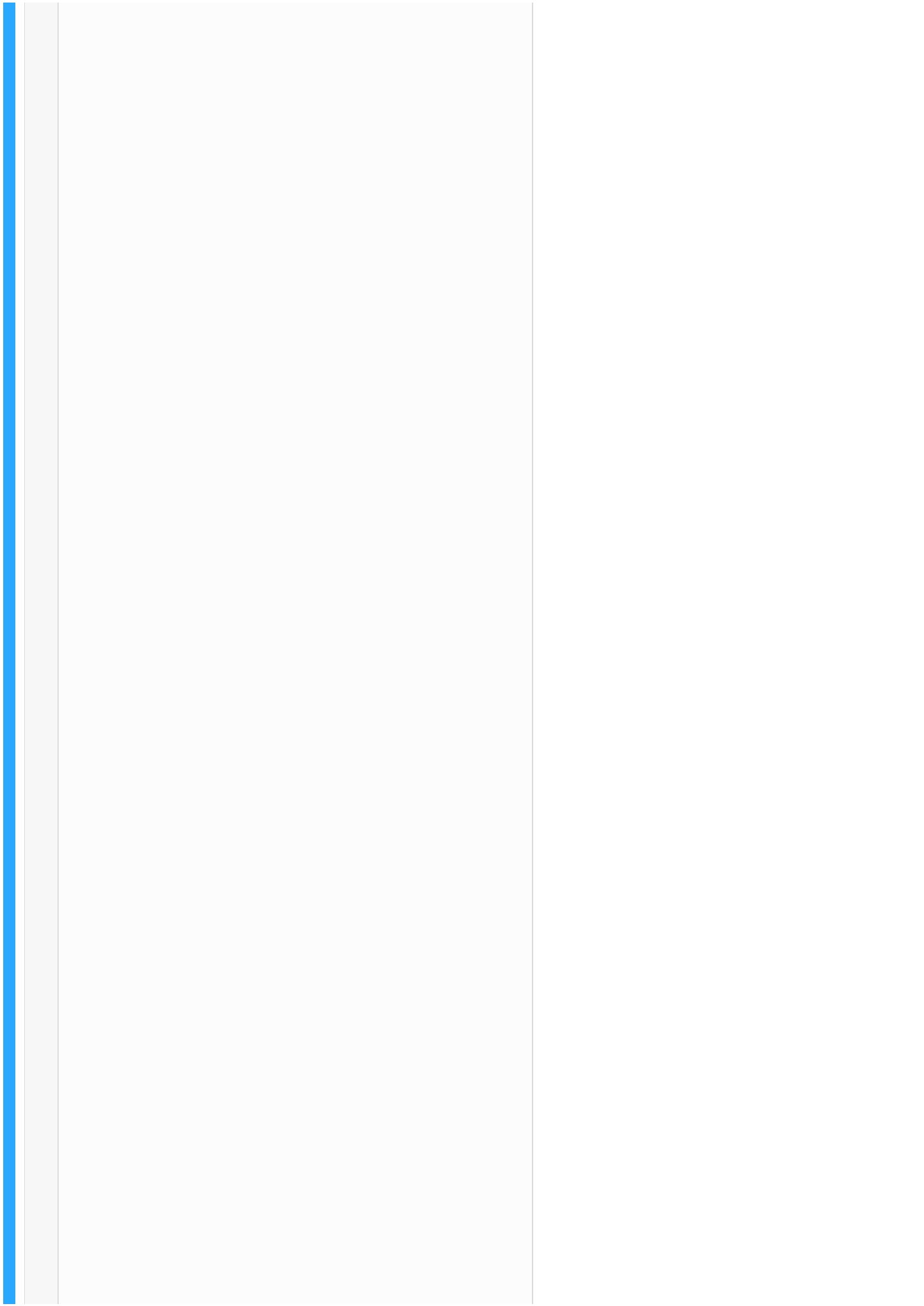
DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)



All in all, it's not too complicated to test a Slim action, and `Environment::mock()` makes it easy to set up particular test cases.

[Power realtime chat, IoT and messaging apps at scale](#). Pubsub realtime messaging, functions, chat, presence, push, notifications, blocks catalog and more.

Like This Article? Read More From DZone



First Beta of Slim Framework 3



Web App Evolution: From Spaghetti to Layered Cake - Part I



Using Eloquent in Slim Framework



[Free DZone Refcard](#)
Low-Code Application Development

Published at DZone with permission of Rob Allen , DZone MVB.
[See the original article here.](#)
Opinions expressed by DZone contributors are their own.

Web Dev Partner Resources

Building & Managing Single Page Apps in your CMS [eBook]

DotCMS

HERE ranks #1 for developers against Google and Mapbox

HERE

ONLYOFFICE code samples: see how editors will work within your app

OnlyOffice

The Ultimate 36-Point Checklist for Choosing a Headless CMS (eBook)

DotCMS

Containers 101: Everything You Need to Know About Containers [eBook]

DotCMS

Welcome to the Cognitive Era - The New Generation of Computing

PubNub

From Brick and Mortar to Bits and Bytes: Why Cognitive Services are Transforming eCommerce

PubNub

HERE topples Google to take first place

HERE

Bring collaborative document editing to your web app with ONLYOFFICE

OnlyOffice

Get 20 location-enabled APIs and SDKs for free with HERE

HERE

A World Transformed: Building Smarter, Next Generation Apps with Cognitive Services

PubNub

Detailed documentation for ONLYOFFICE integration

OnlyOffice

React Query Builder With Cube.js

dotcms

In this post, we look at how we can use these two open source libraries to create a query functionality in an application using JavaScript.



by Artyom Keydunov · Mar 28, 19 · Web Dev
Zone · Tutorial

Like (1) Comment (0) Save

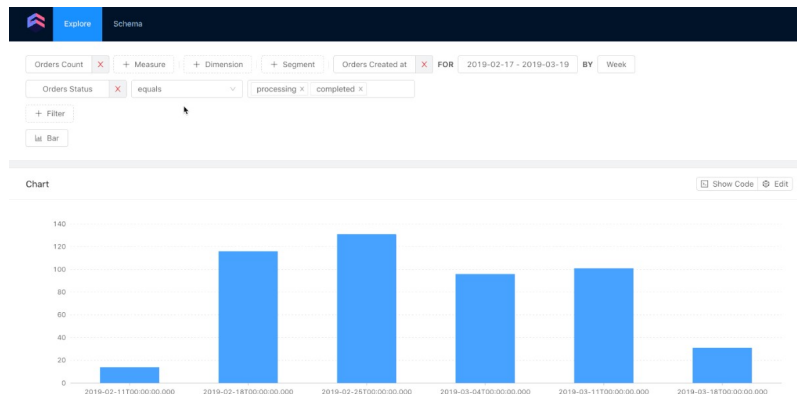
Tweet

662 Views

A true open source, API-first CMS — giving you the power to think outside the webpage. [Try it for free.](#)

Starting from version 0.4, the React [Cube.js](#) client comes with the `<QueryBuilder />` component. It is designed to help developers build interactive analytics query builders. The `<QueryBuilder />` abstracts state management and API calls to the Cube.js backend. It uses the [render prop](#) and doesn't render anything itself, but calls the render function instead. This way it gives maximum flexibility to building a custom-tailored UI with a minimal API.

The example below shows the `<QueryBuilder />` component in action with Ant Design UI framework elements.



The above example is from Cube.js Playground. [You can check its source code on GitHub.](#)

This tutorial walks through building the much simpler version of the query builder. But it covers all the basics you need to build one of your own.

Setup a Demo Backend

If you already have Cube.js backend up and running you can skip this step.

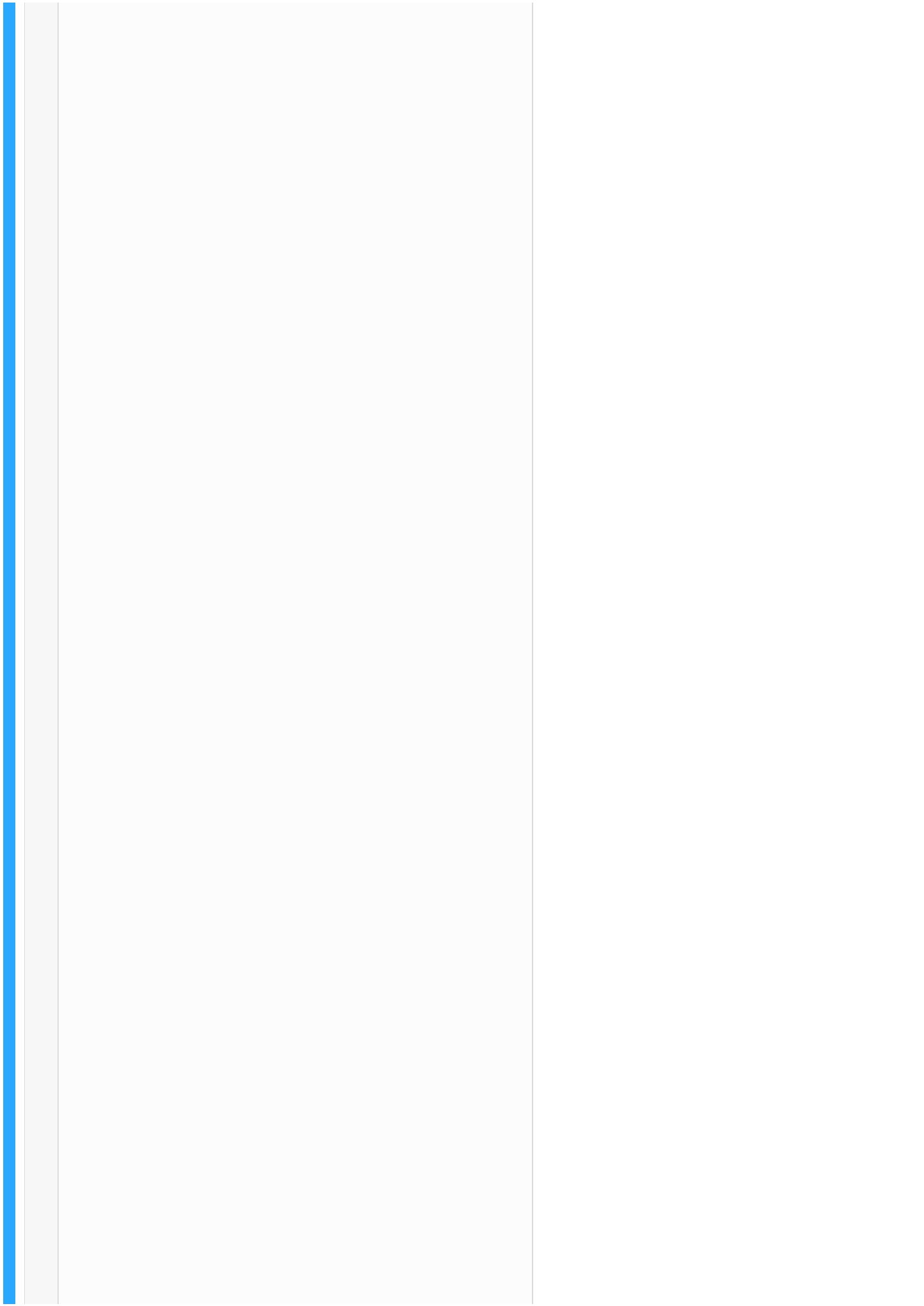
First, let's install the Cube.js CLI and create a new application with a Postgres database.

```
1 $ npm install -g cubejs-cli
2 $ cubejs create -d postgres react-query-builder
```





























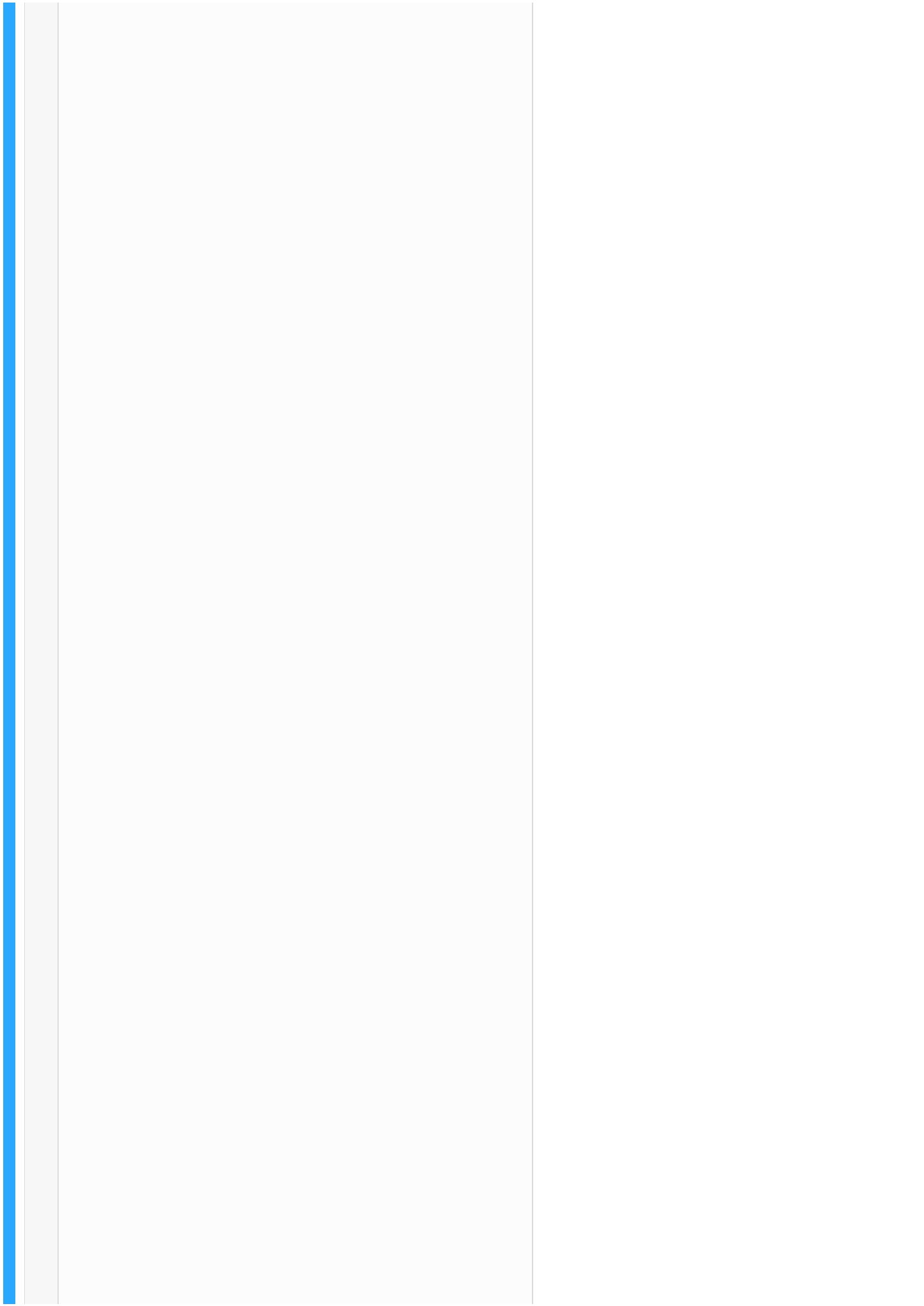








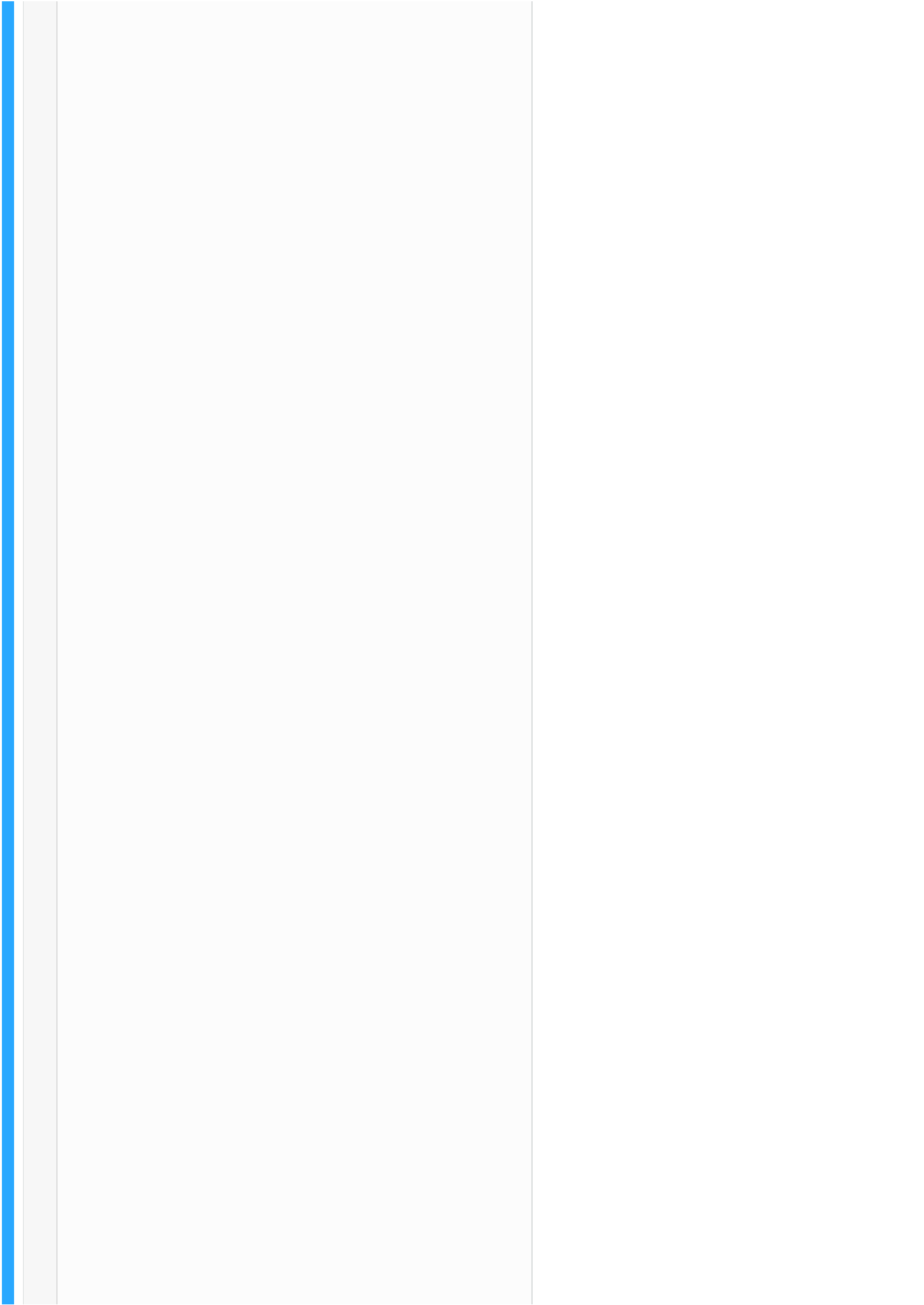
















We host a dump with sample data for tutorials. It is a simple “E-commerce database” with orders, products, product categories, and users tables.

```
1 $ curl http://cube.dev/downloads/ecom-dump.sql >
   ecom-dump.sql
2 $ createdb ecom
3 $ psql --dbname ecom -f ecom-dump.sql
```











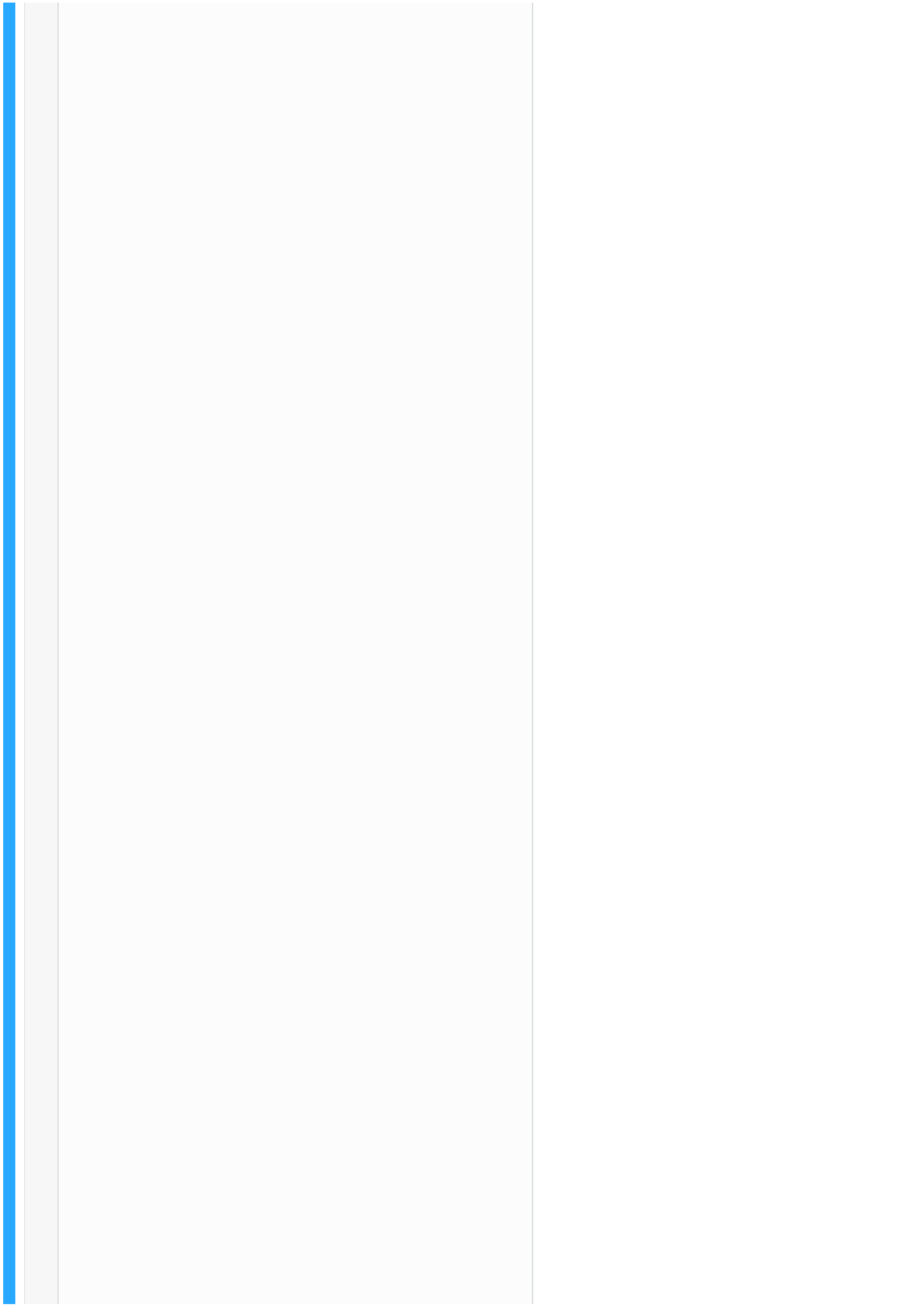








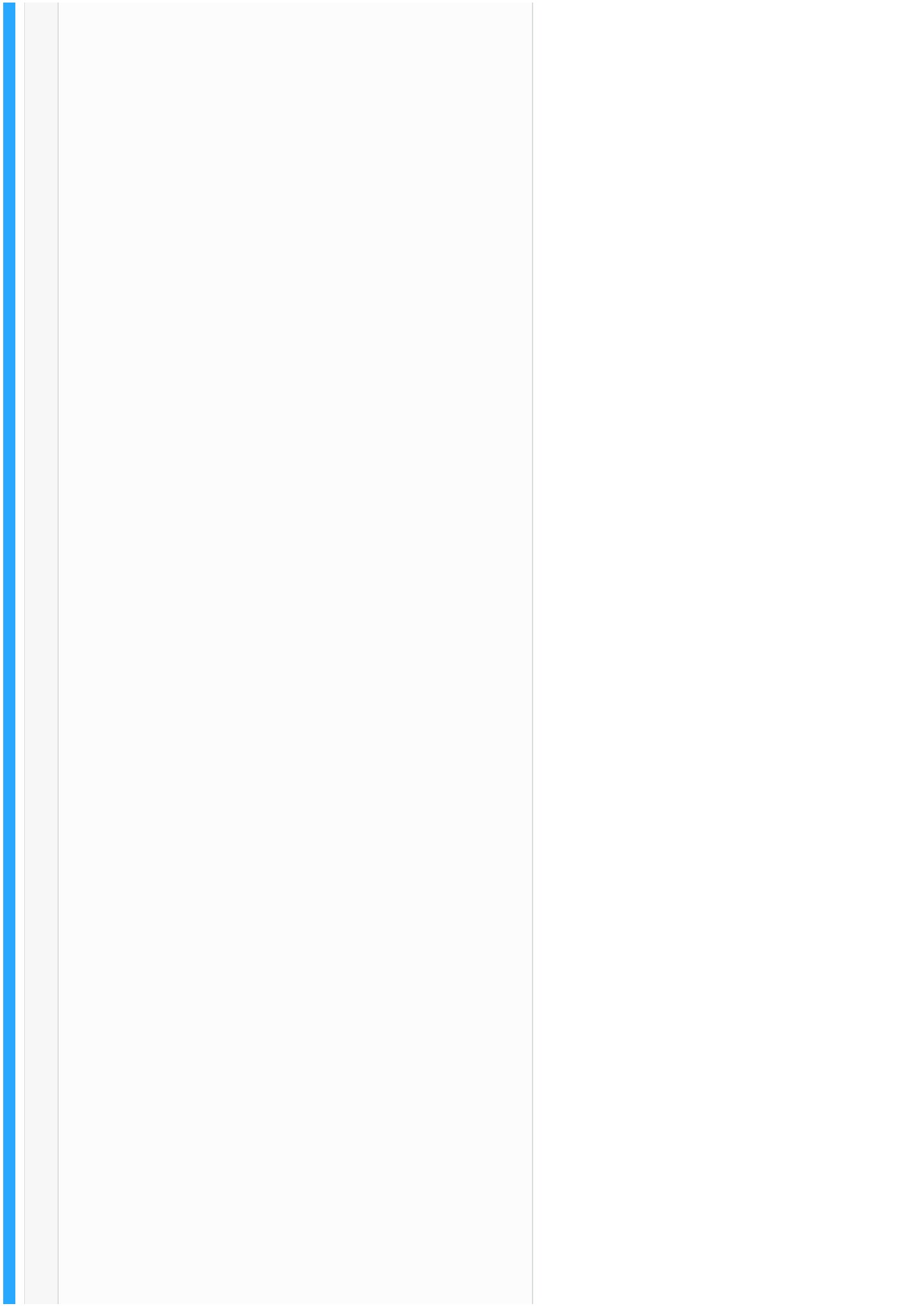






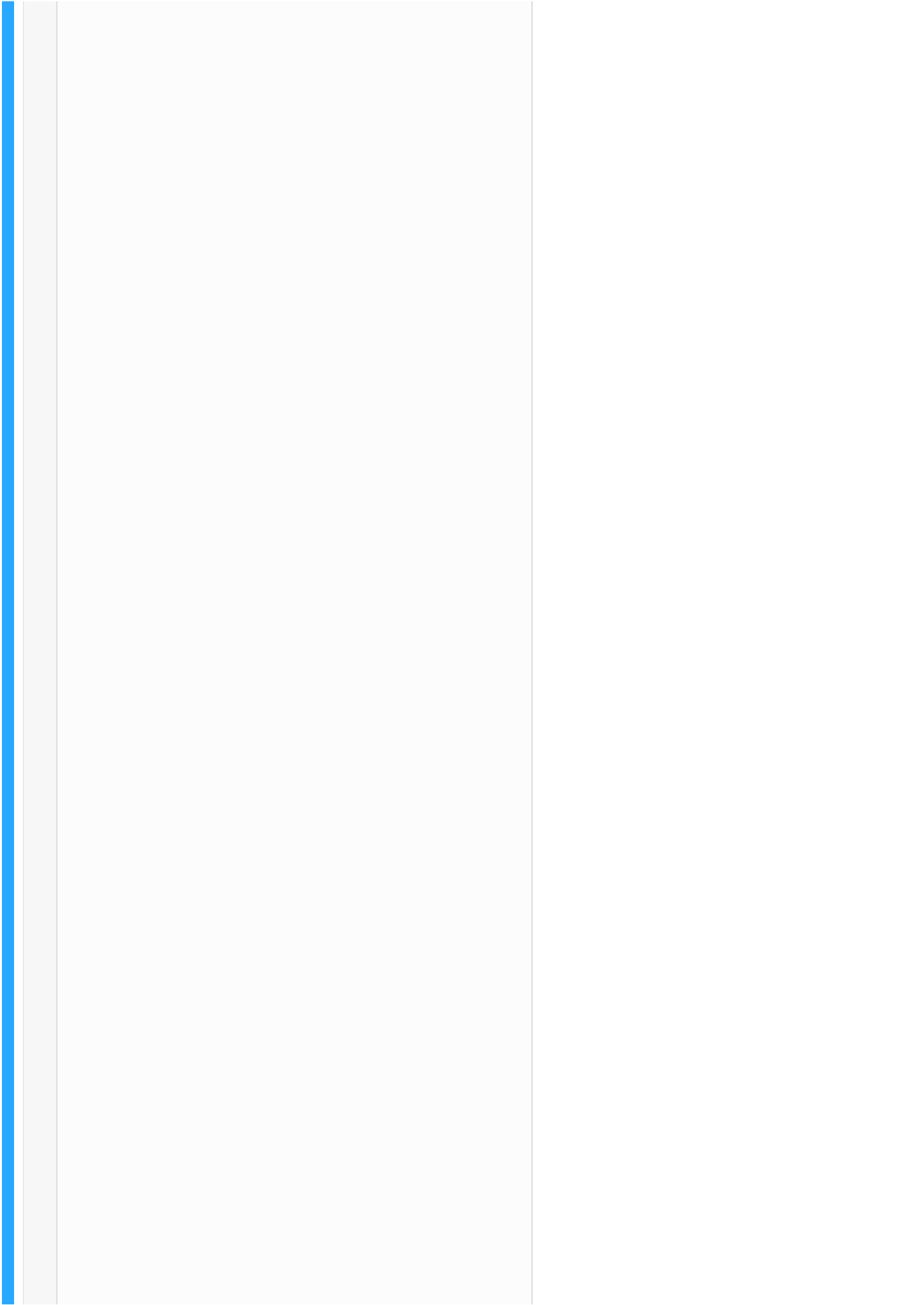


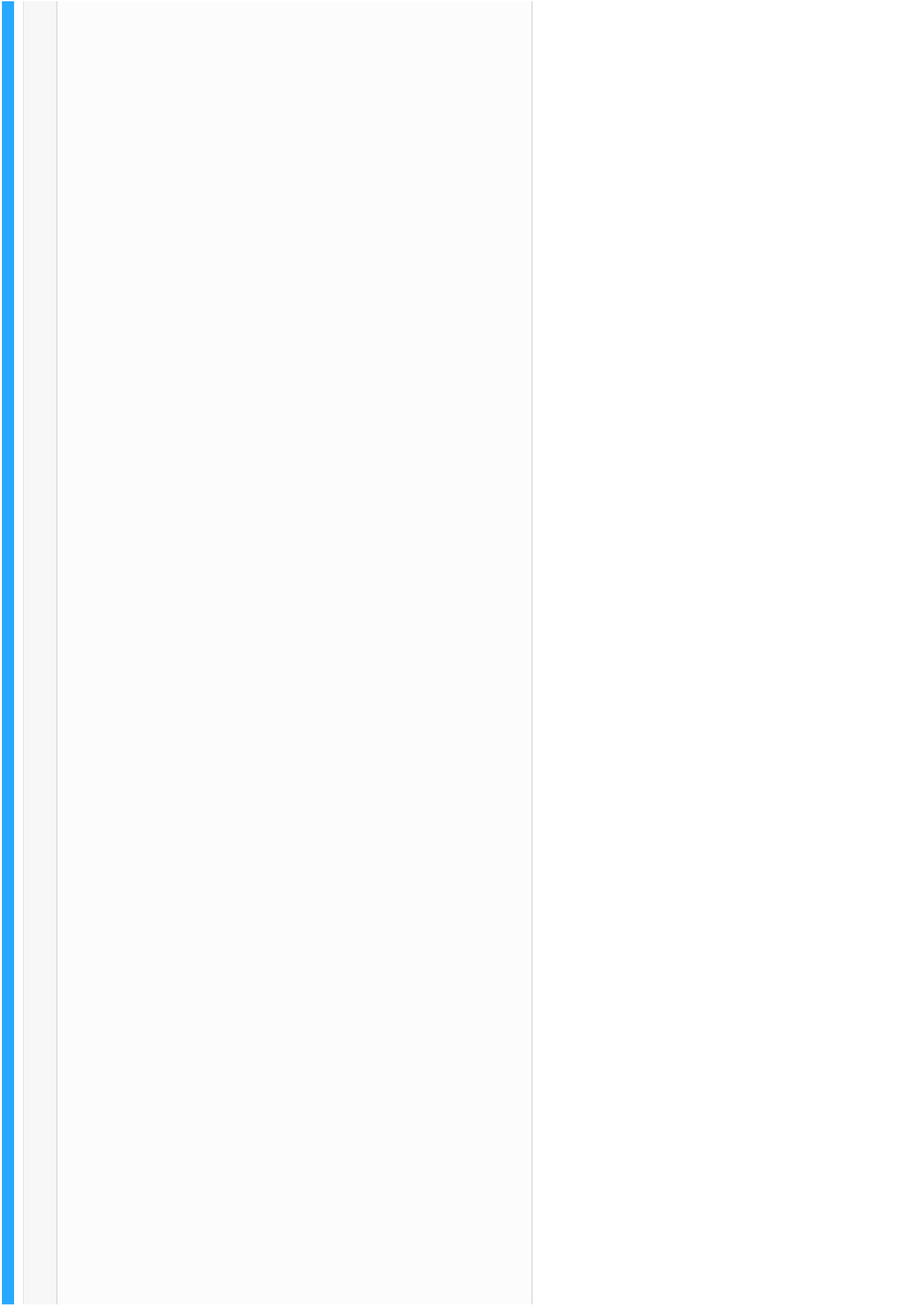


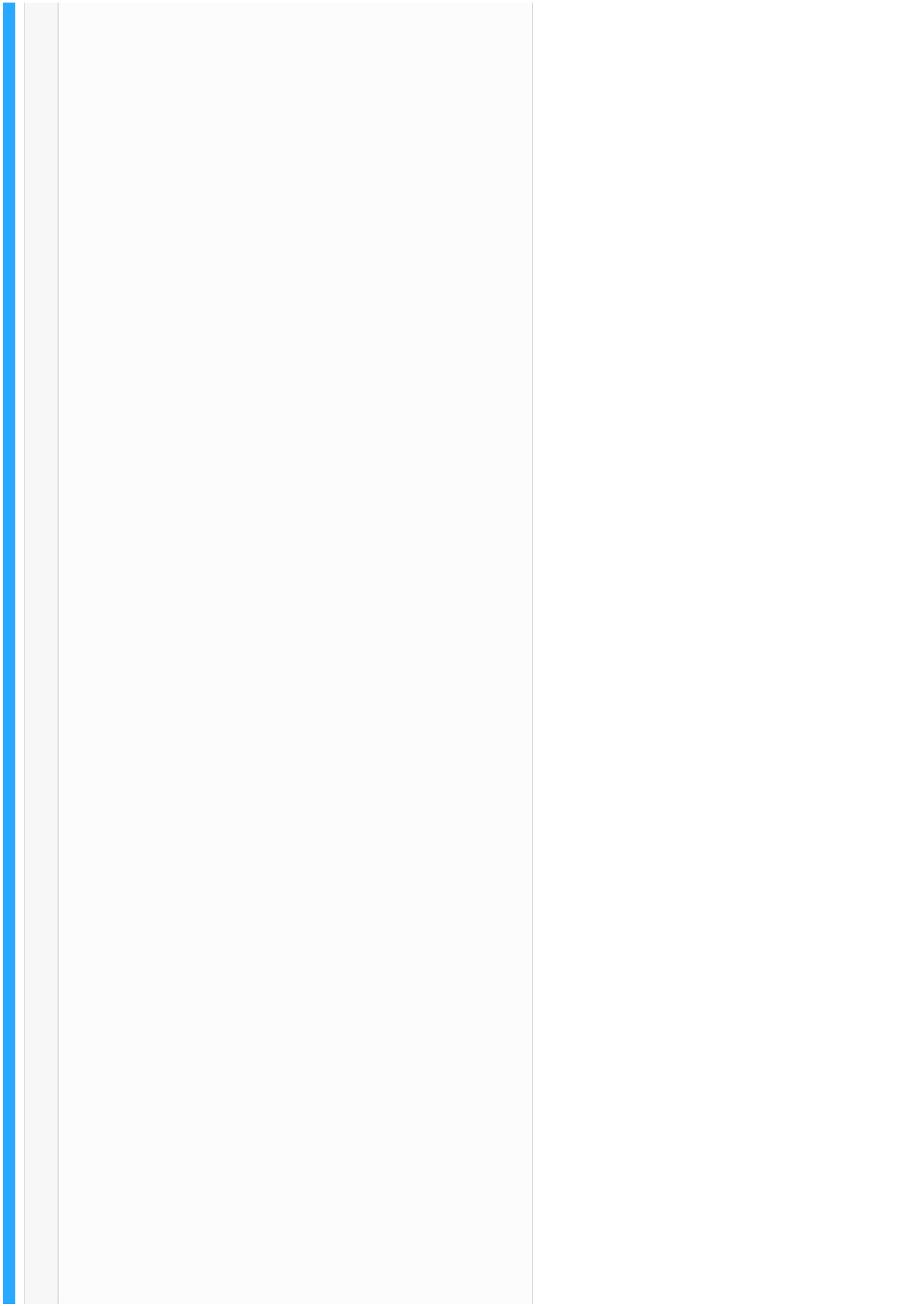




















DZone

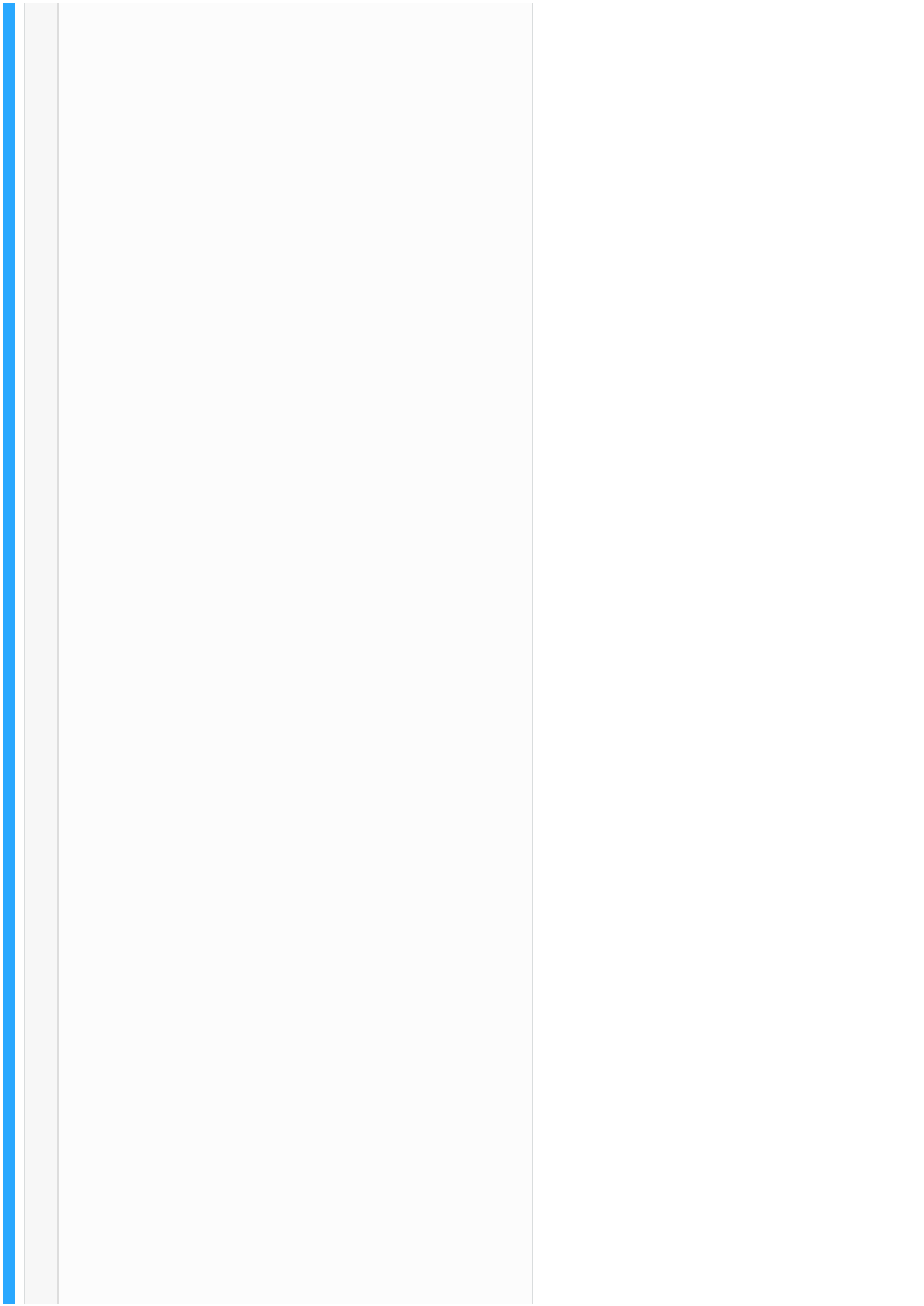
Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)

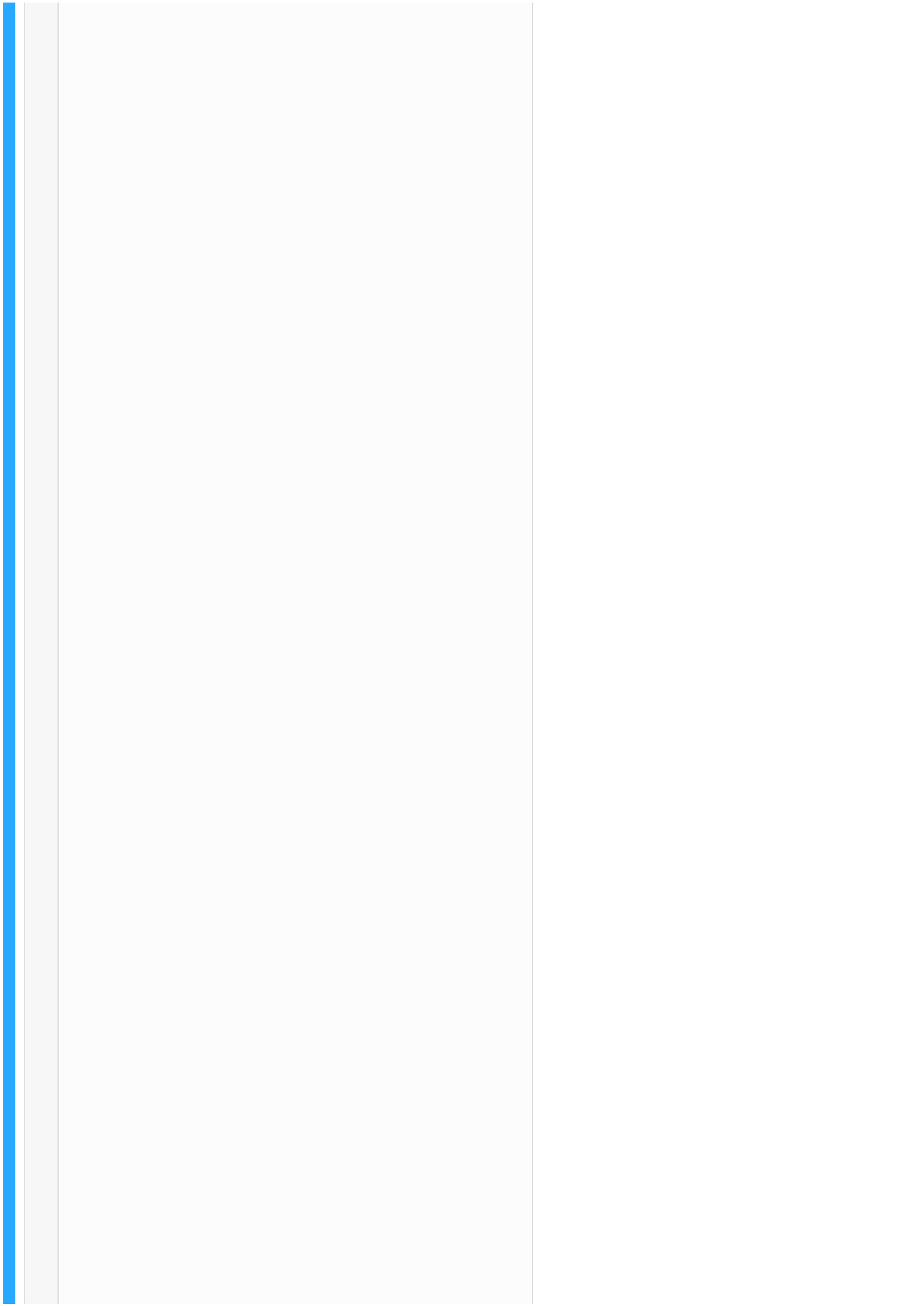


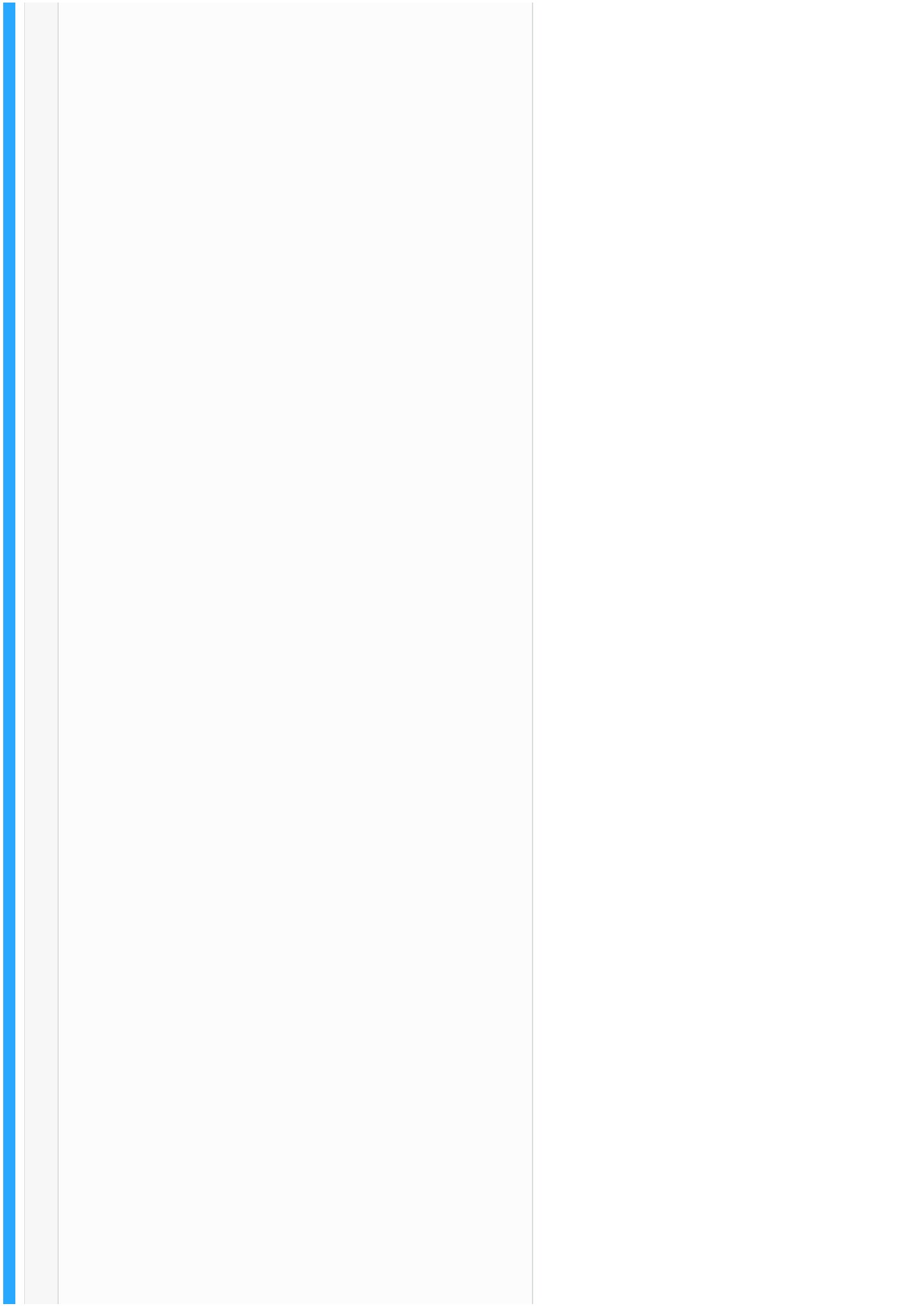




Once you have data in your database, change the content of the `.env` file inside your Cube.js directory to the following. It sets the credentials to access the database, as well as a secret to generate auth tokens.

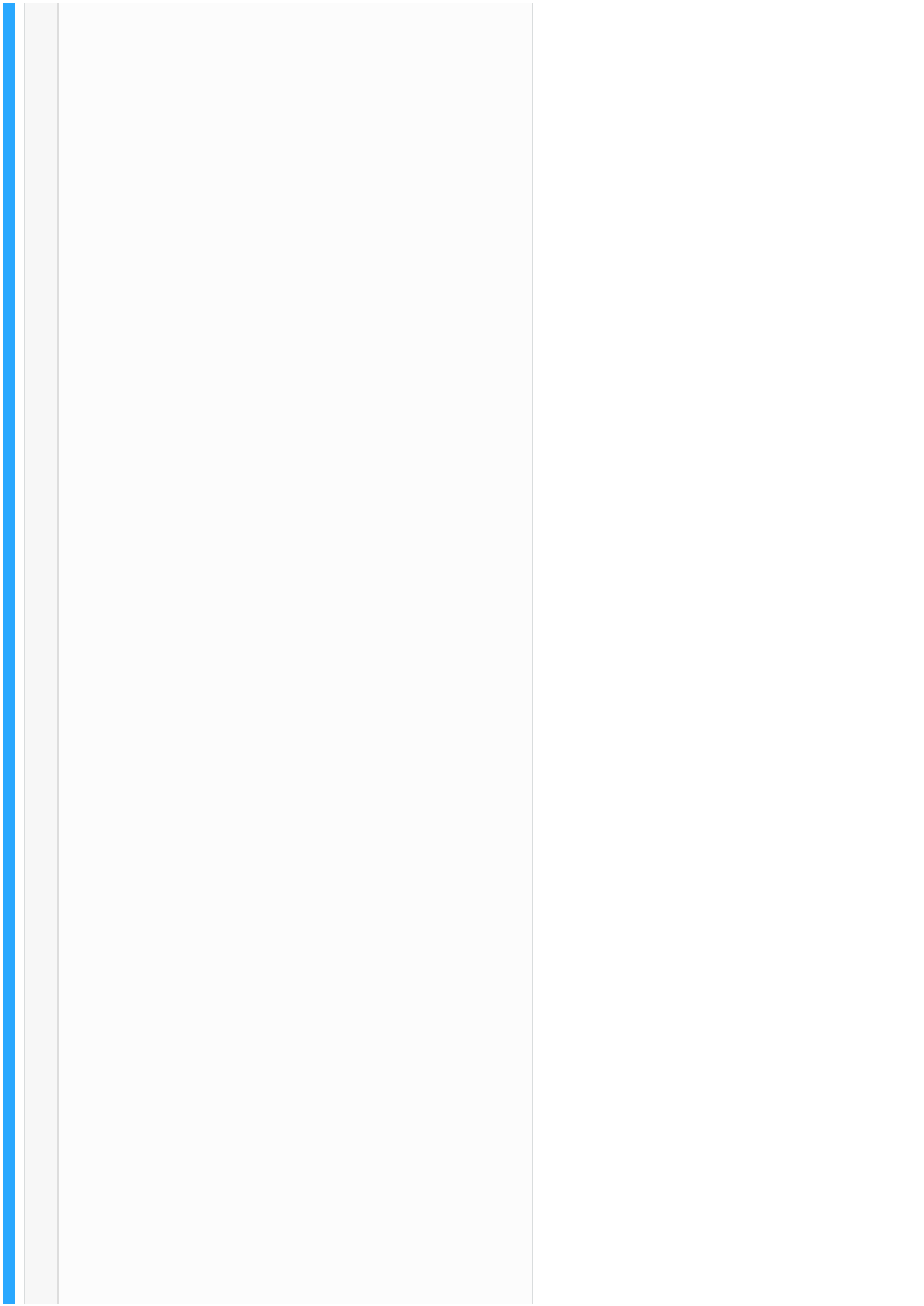
```
1 CUBEJS_DB_NAME=ecom
2 CUBEJS_DB_TYPE=postgres
3 CUBEJS_API_SECRET=SECRET
```









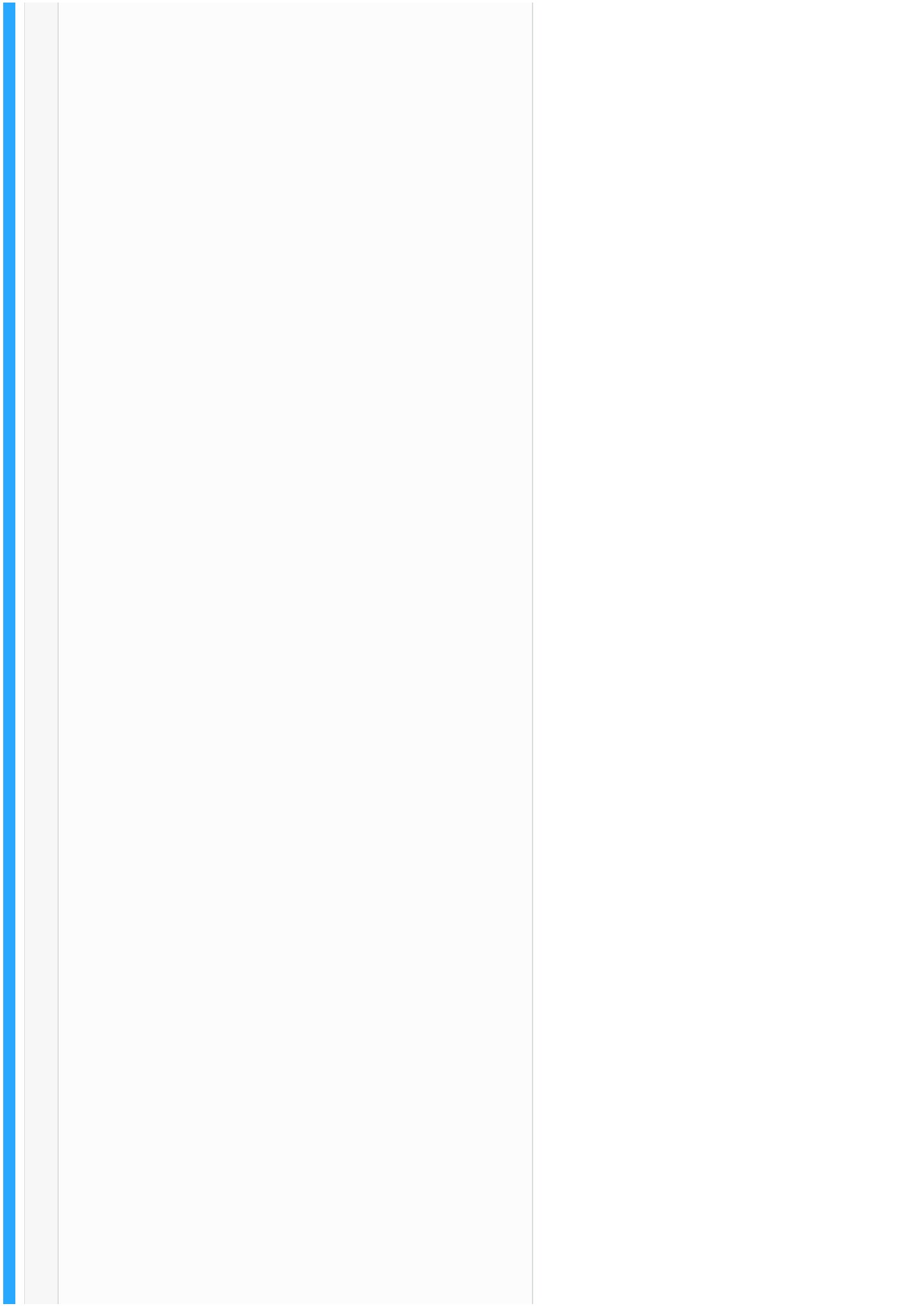


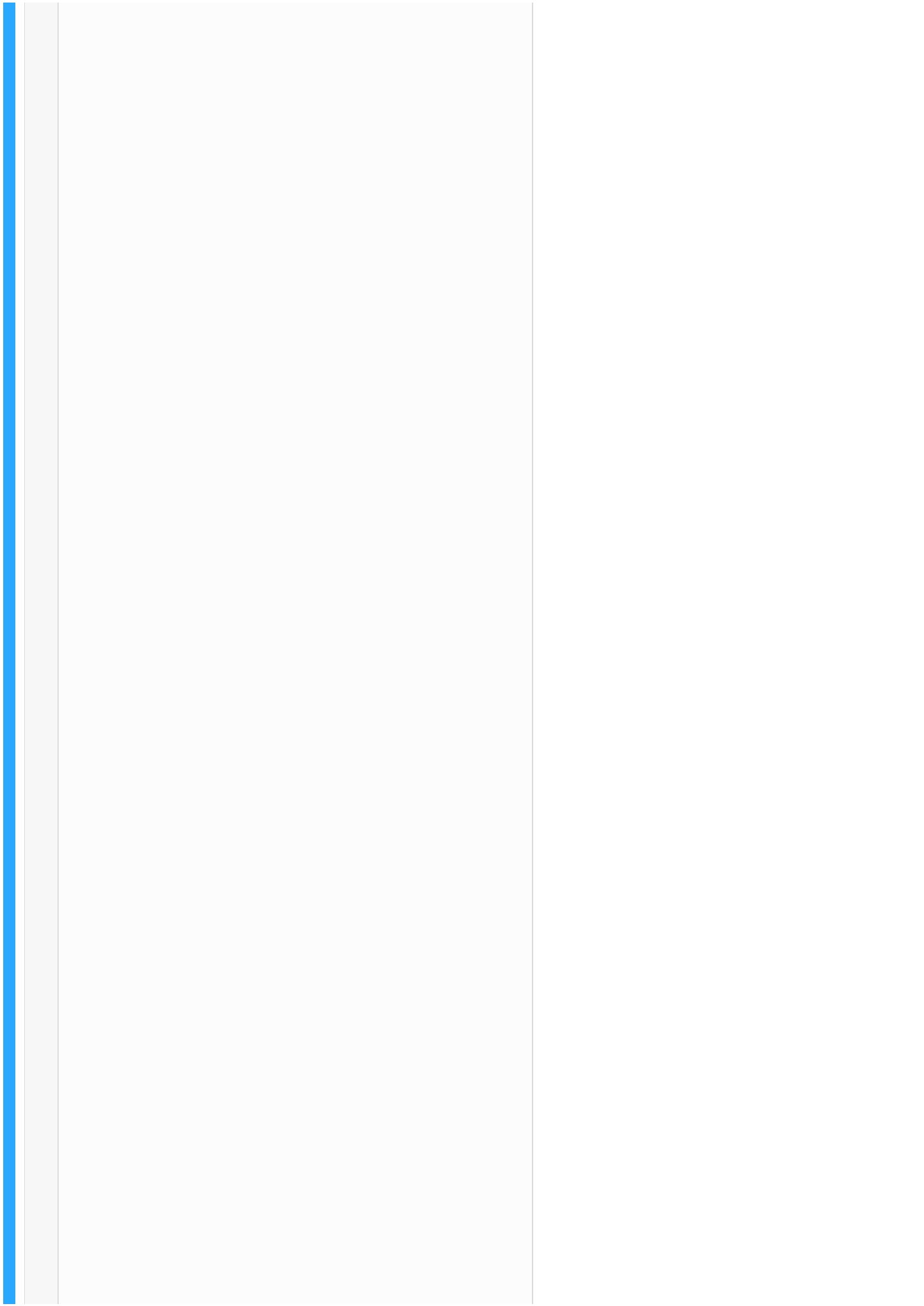


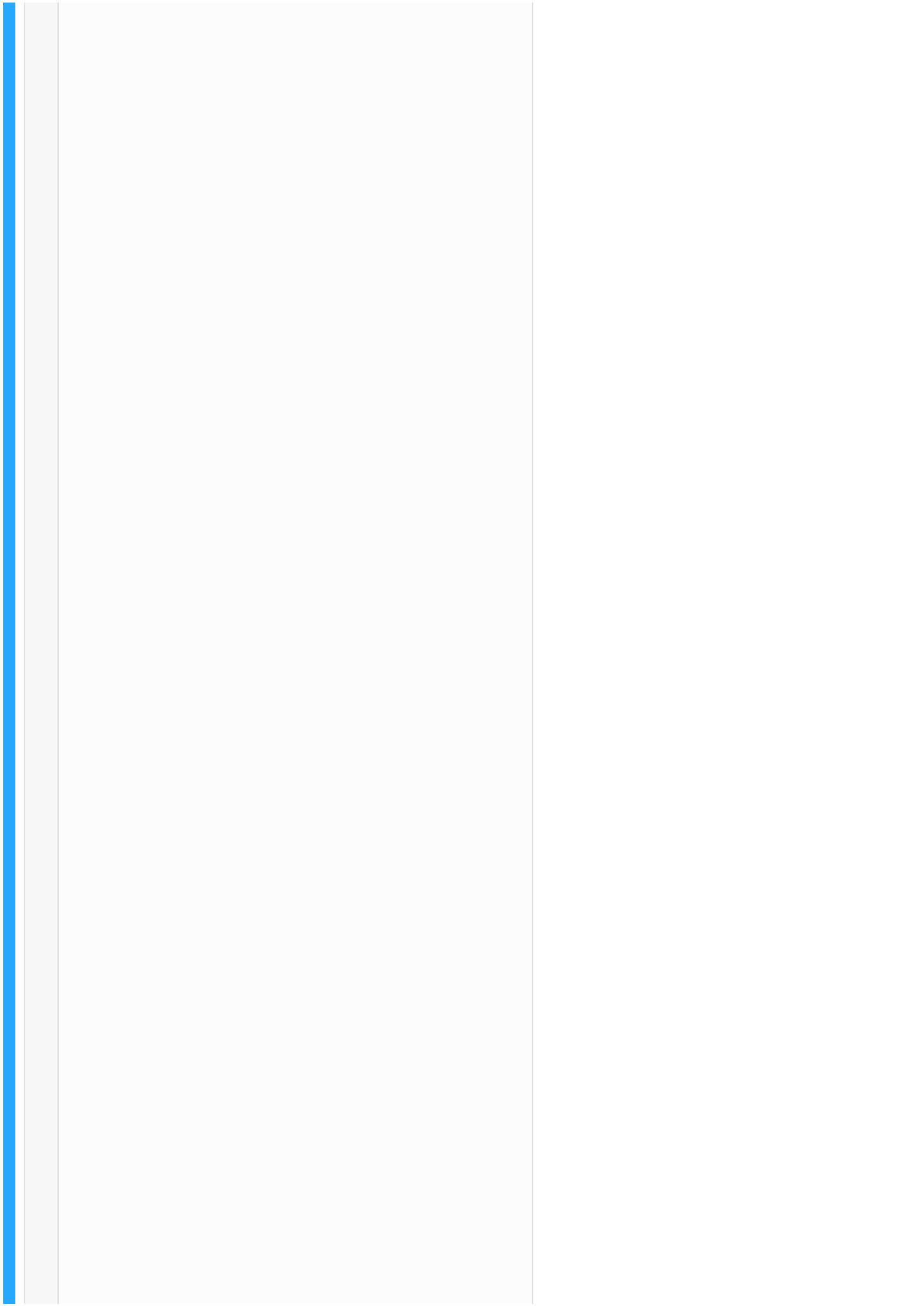










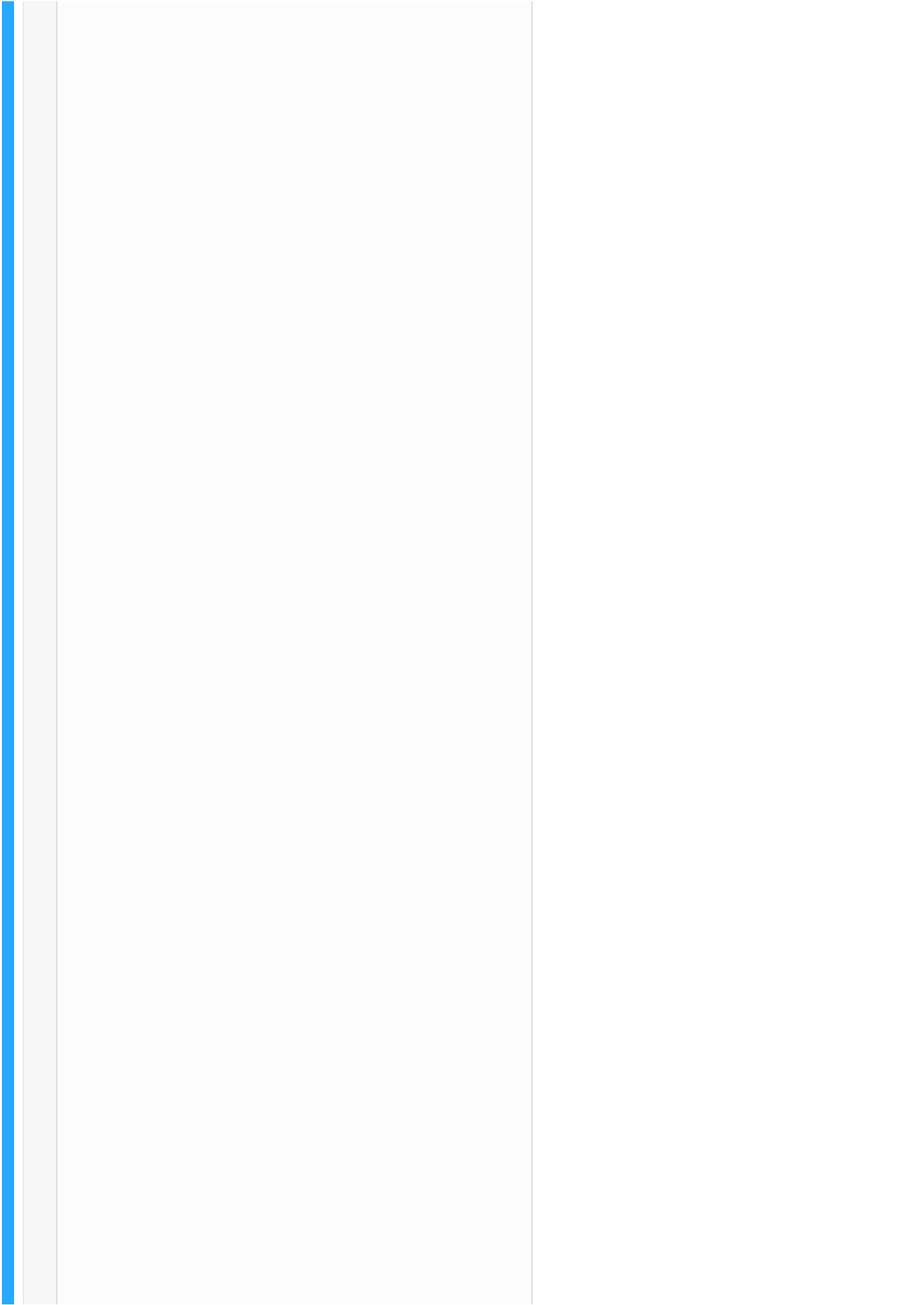








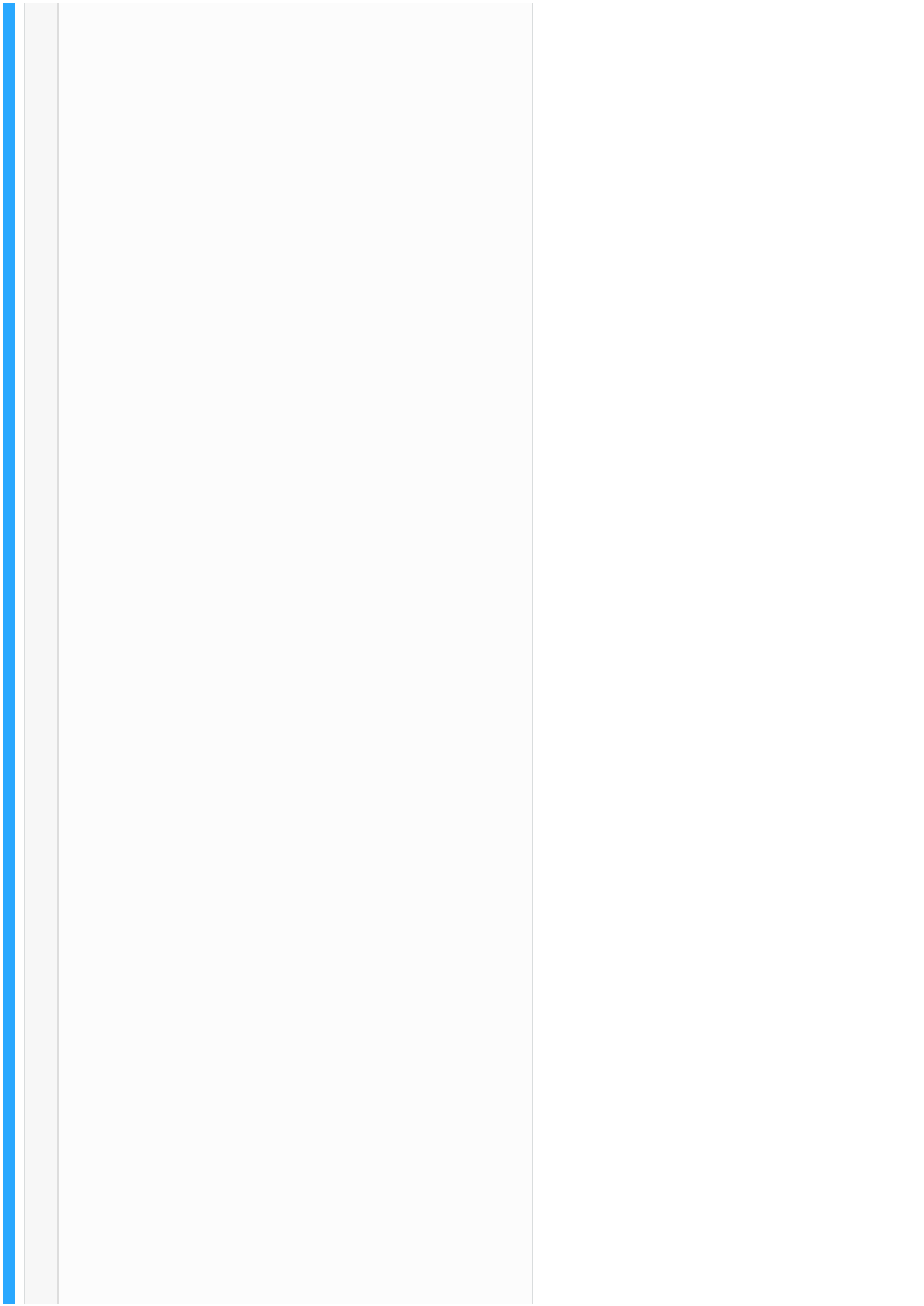


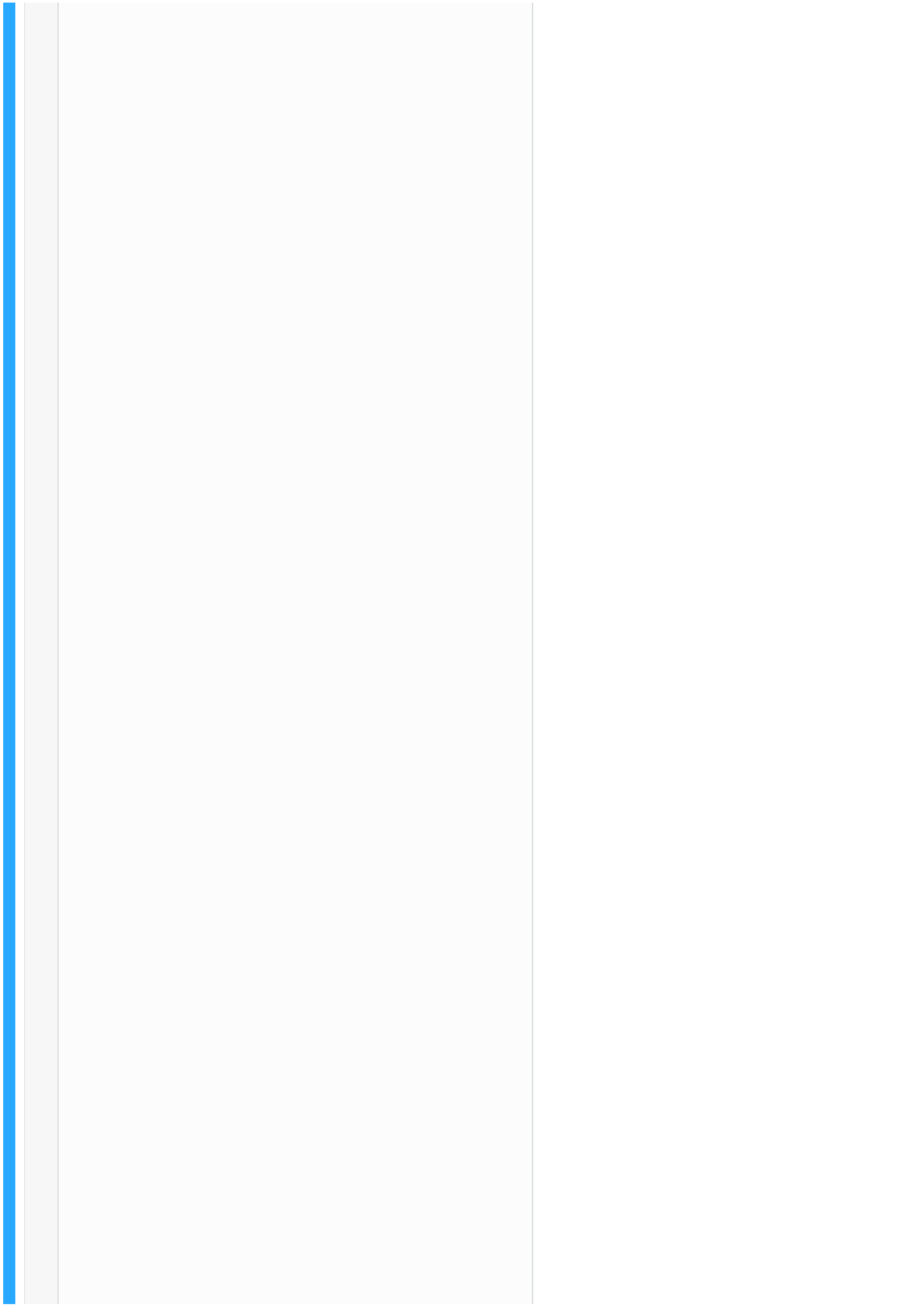












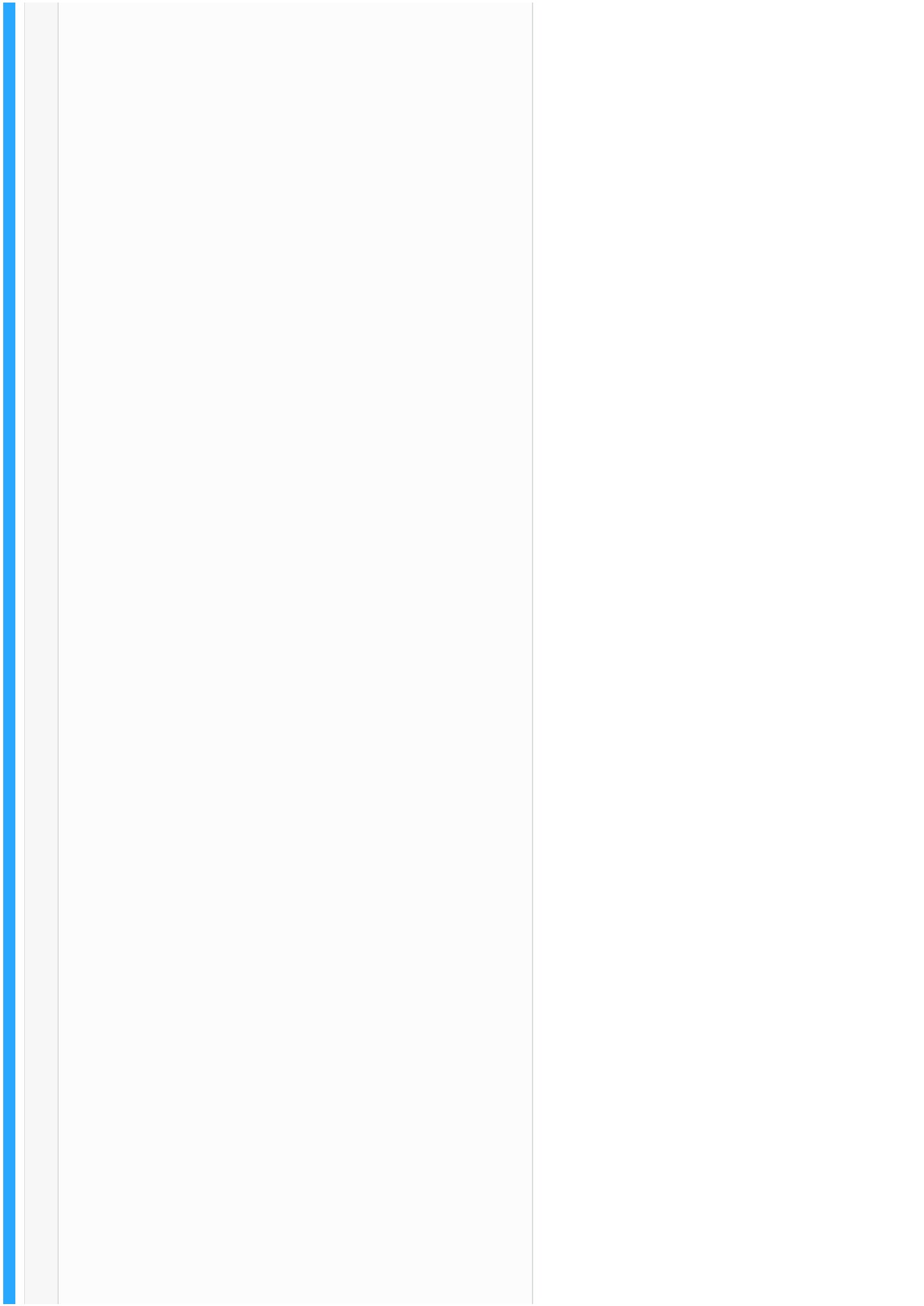


DZone Web Dev Zone

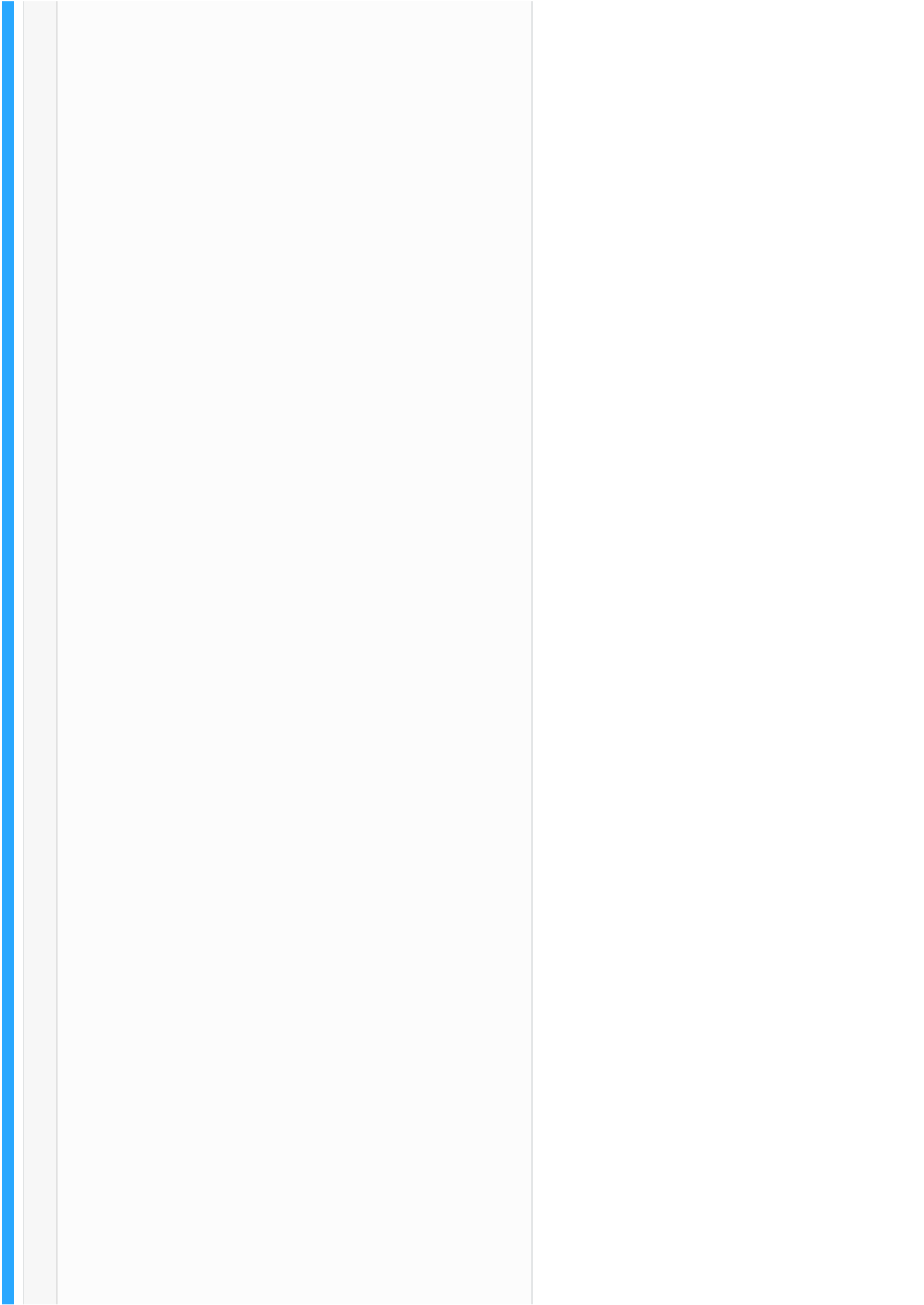
[Log In](#) / [Sign Up](#) 

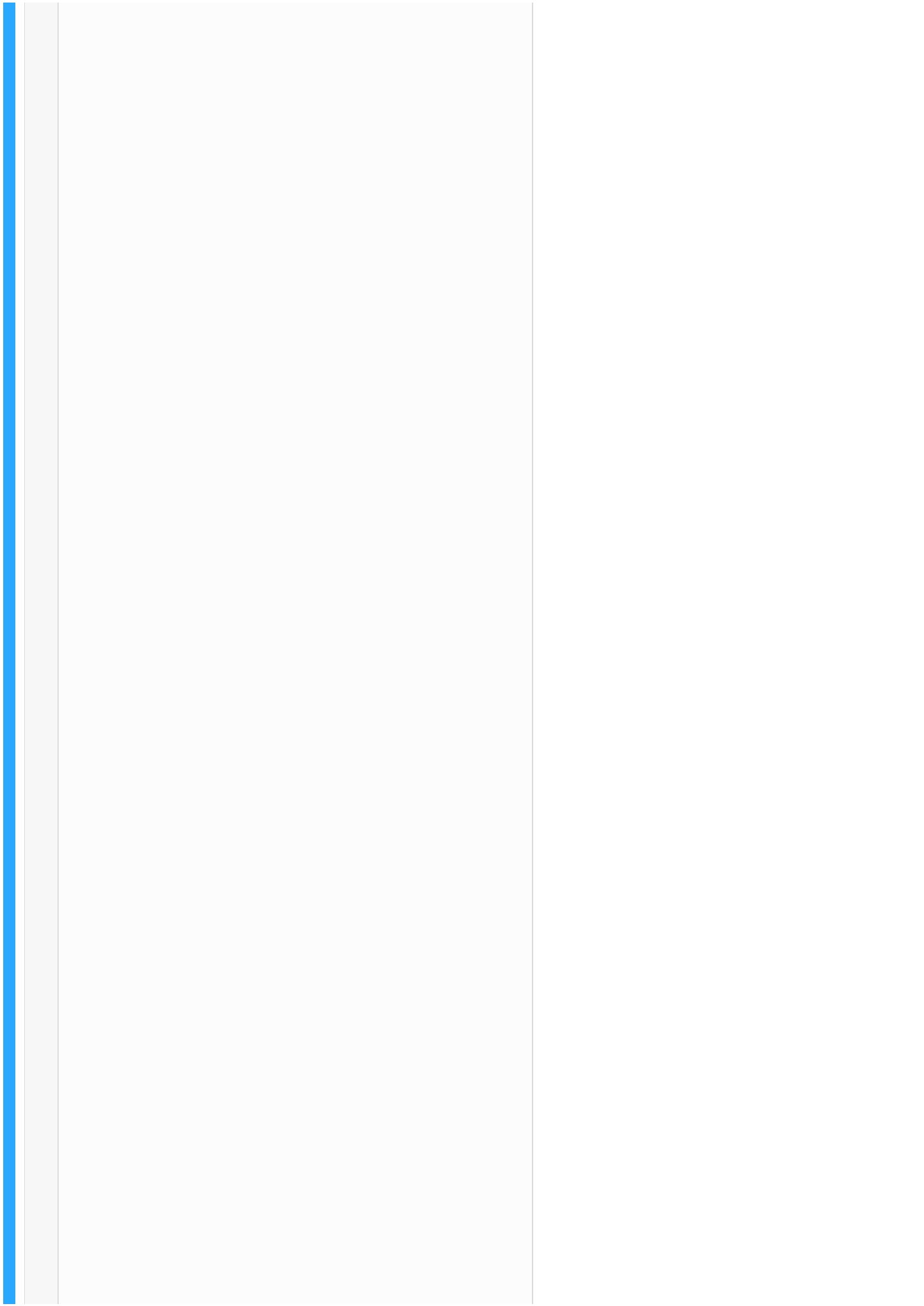


[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)





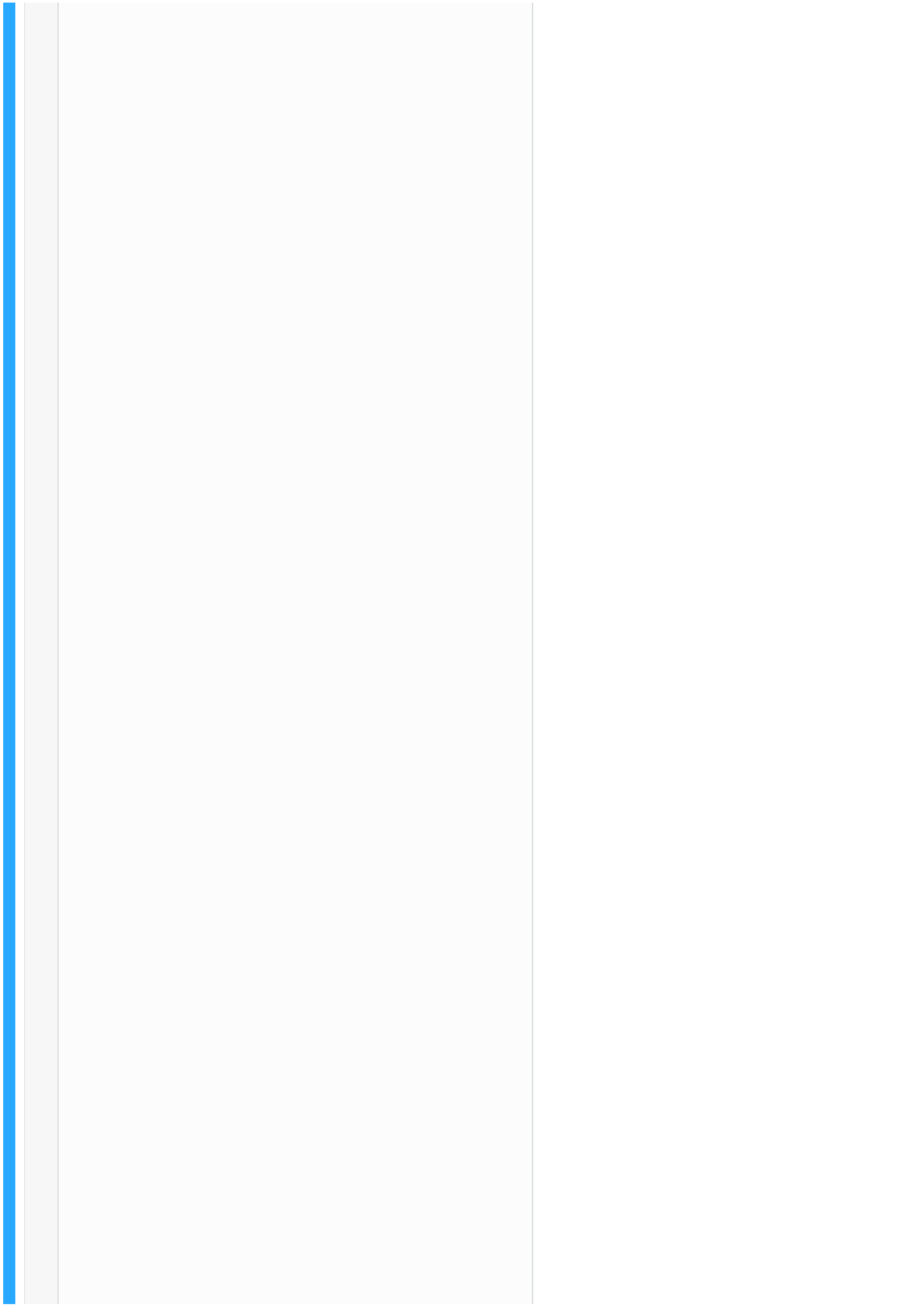


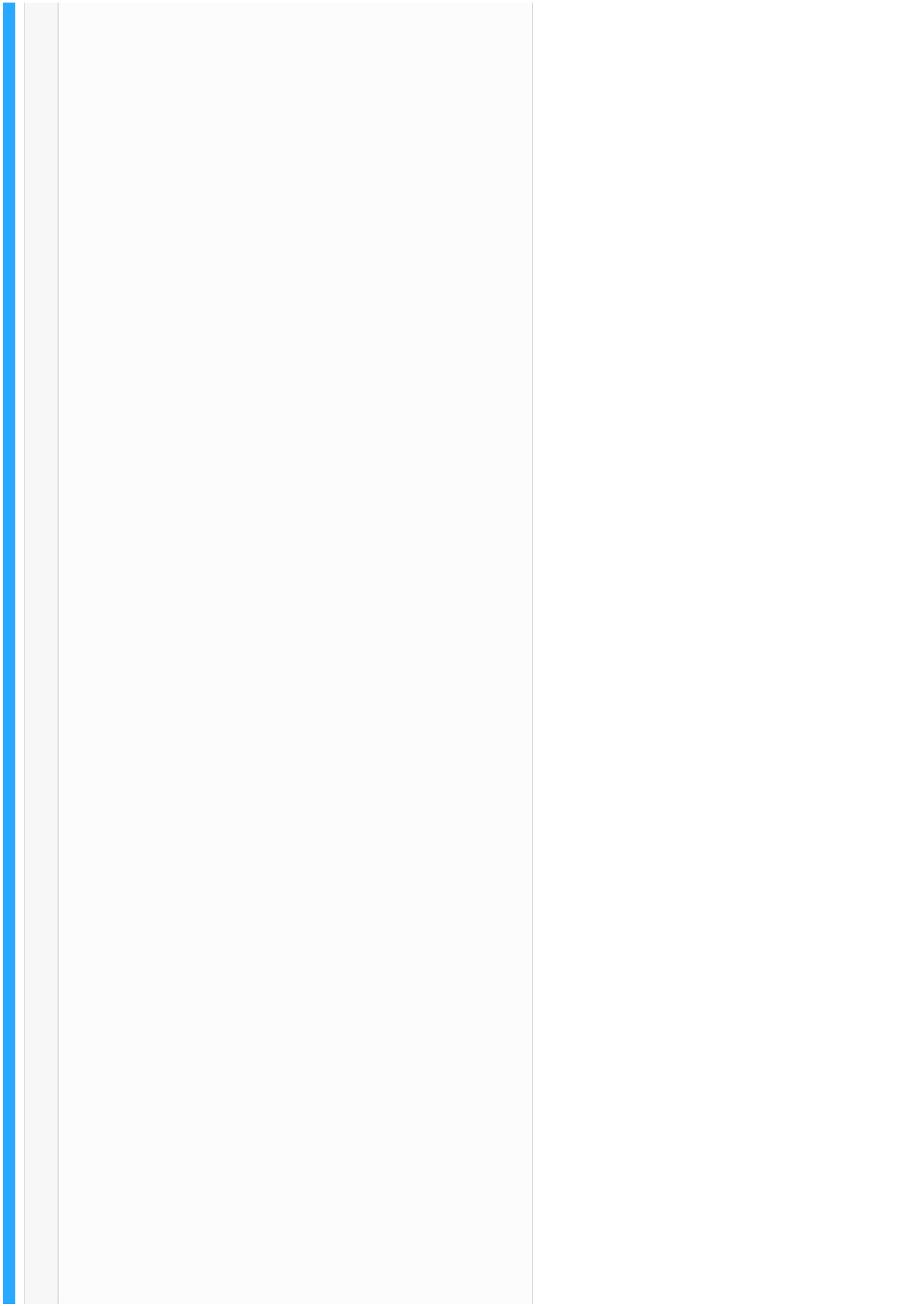


Now that we have everything configured, the last step is to generate a [Cube.js schema](#) based on some of our tables and start the dev server.

```
1 $ cubejs generate -t line_items
2 $ yarn dev
```



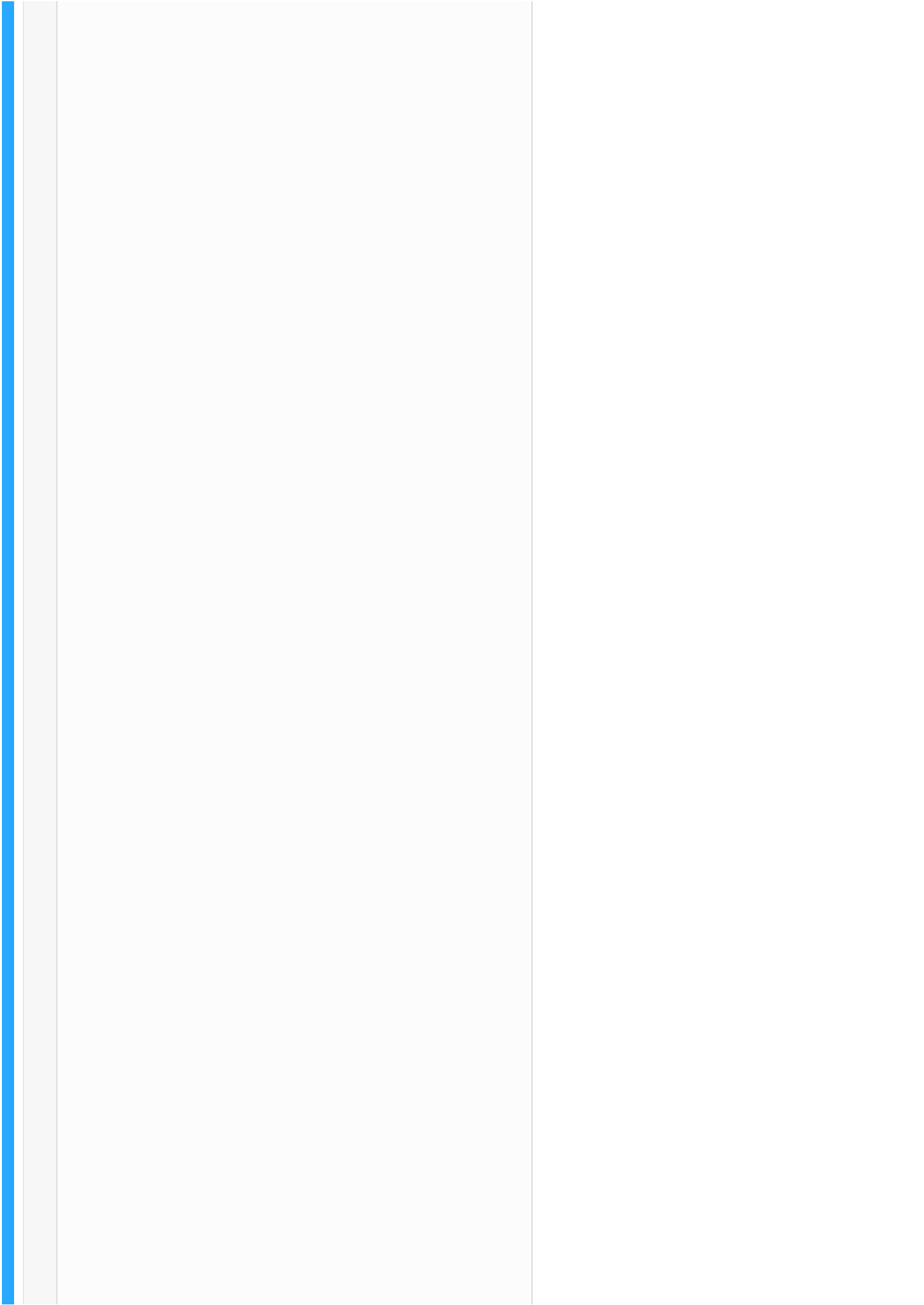




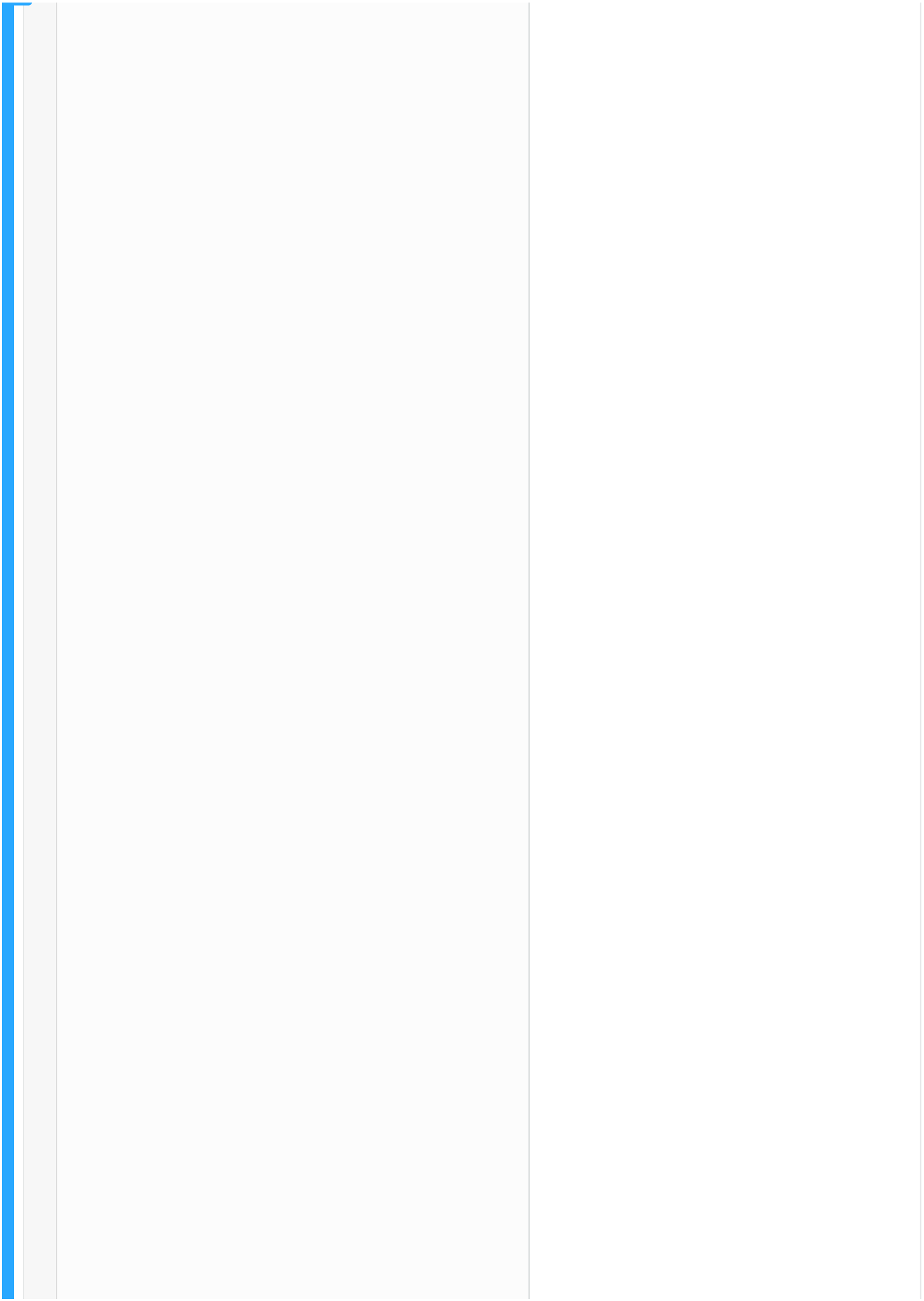


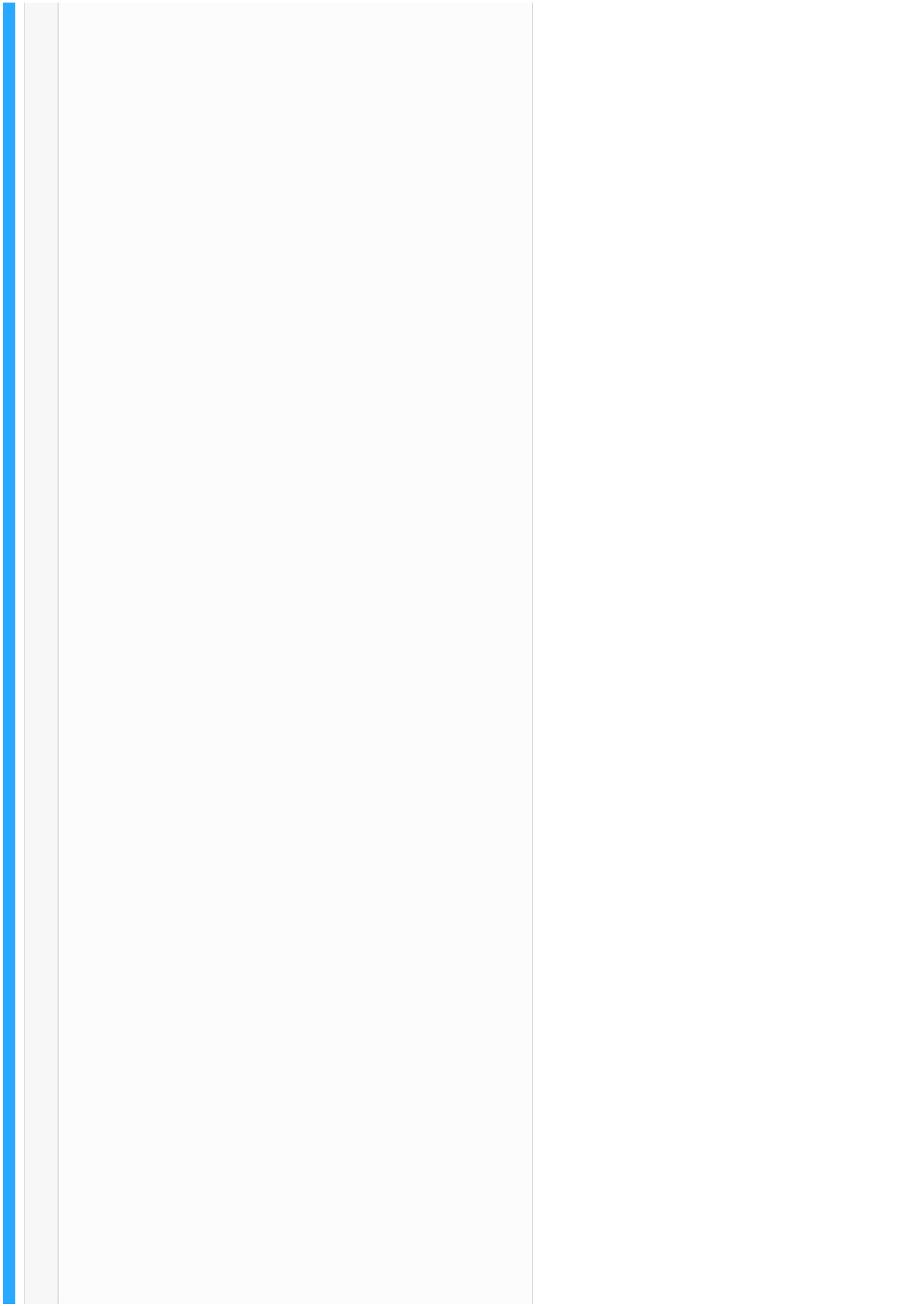








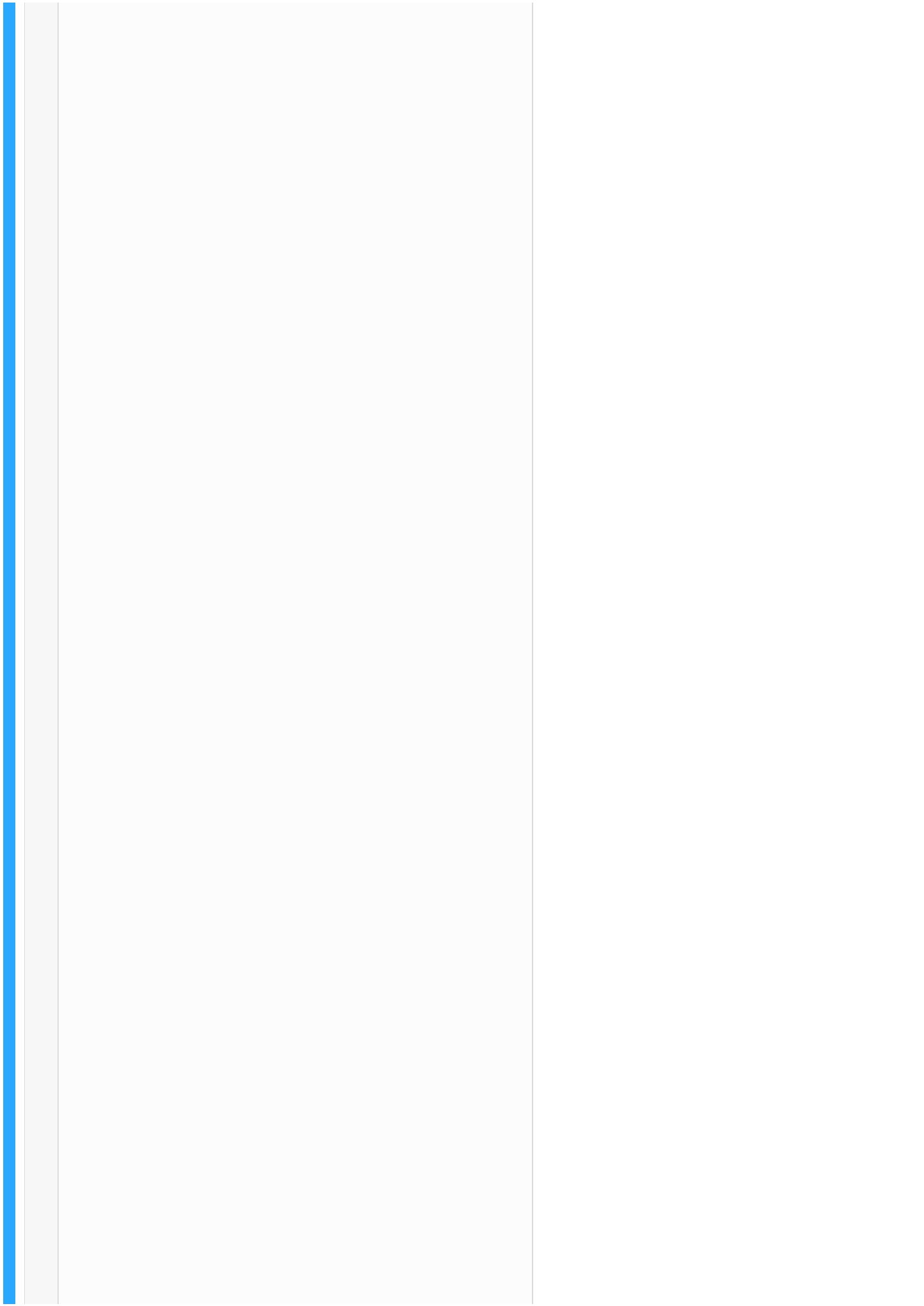


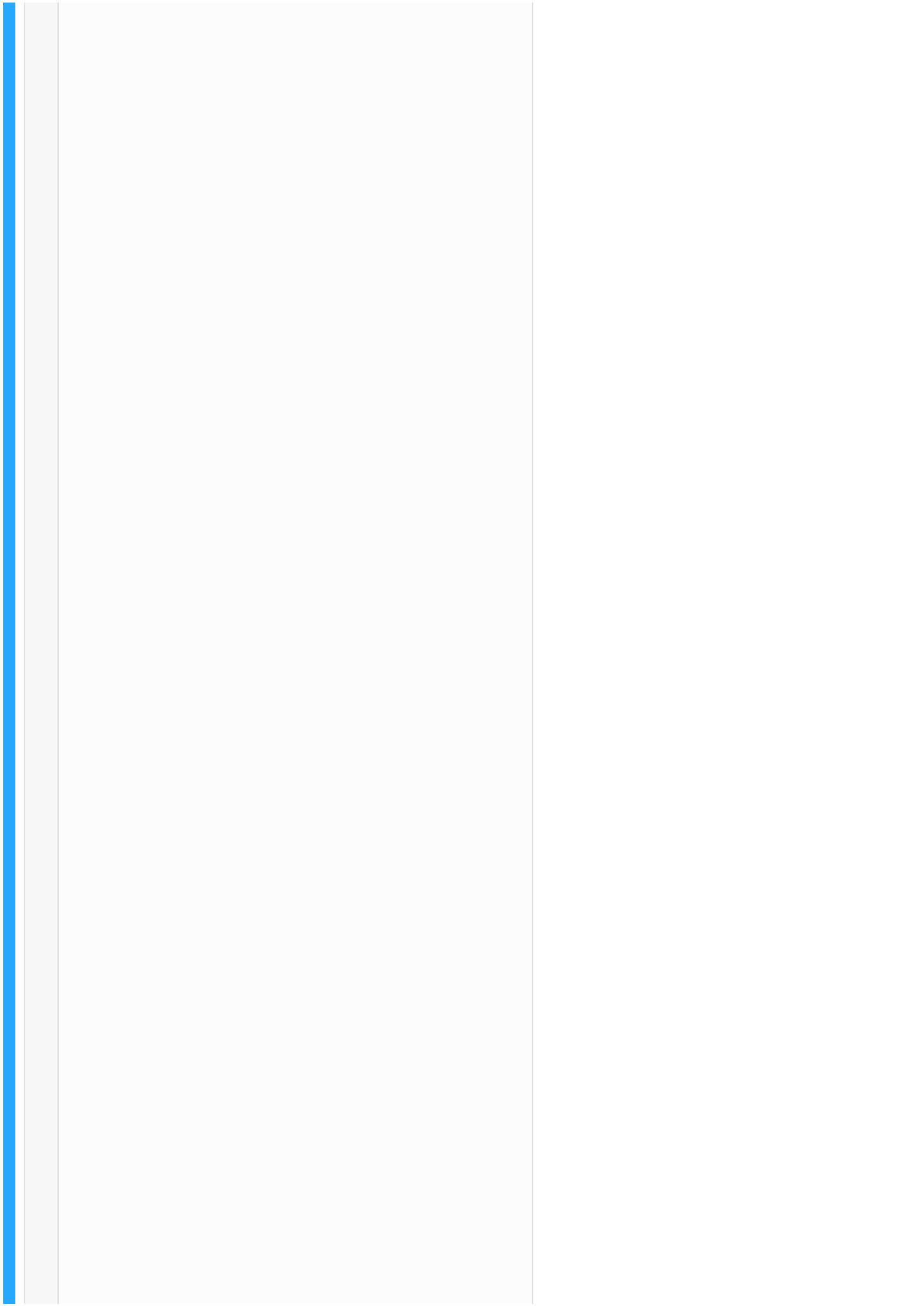


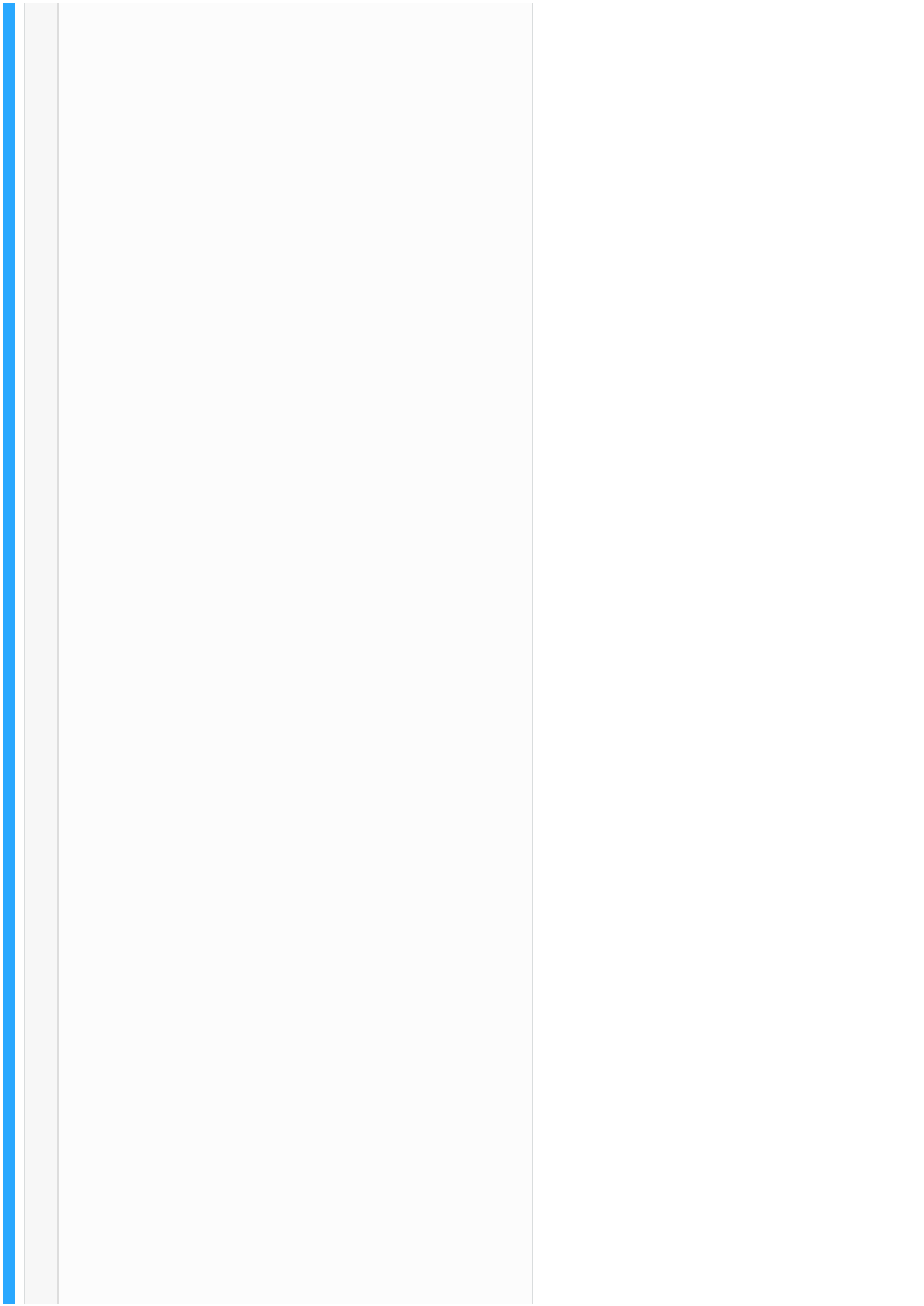






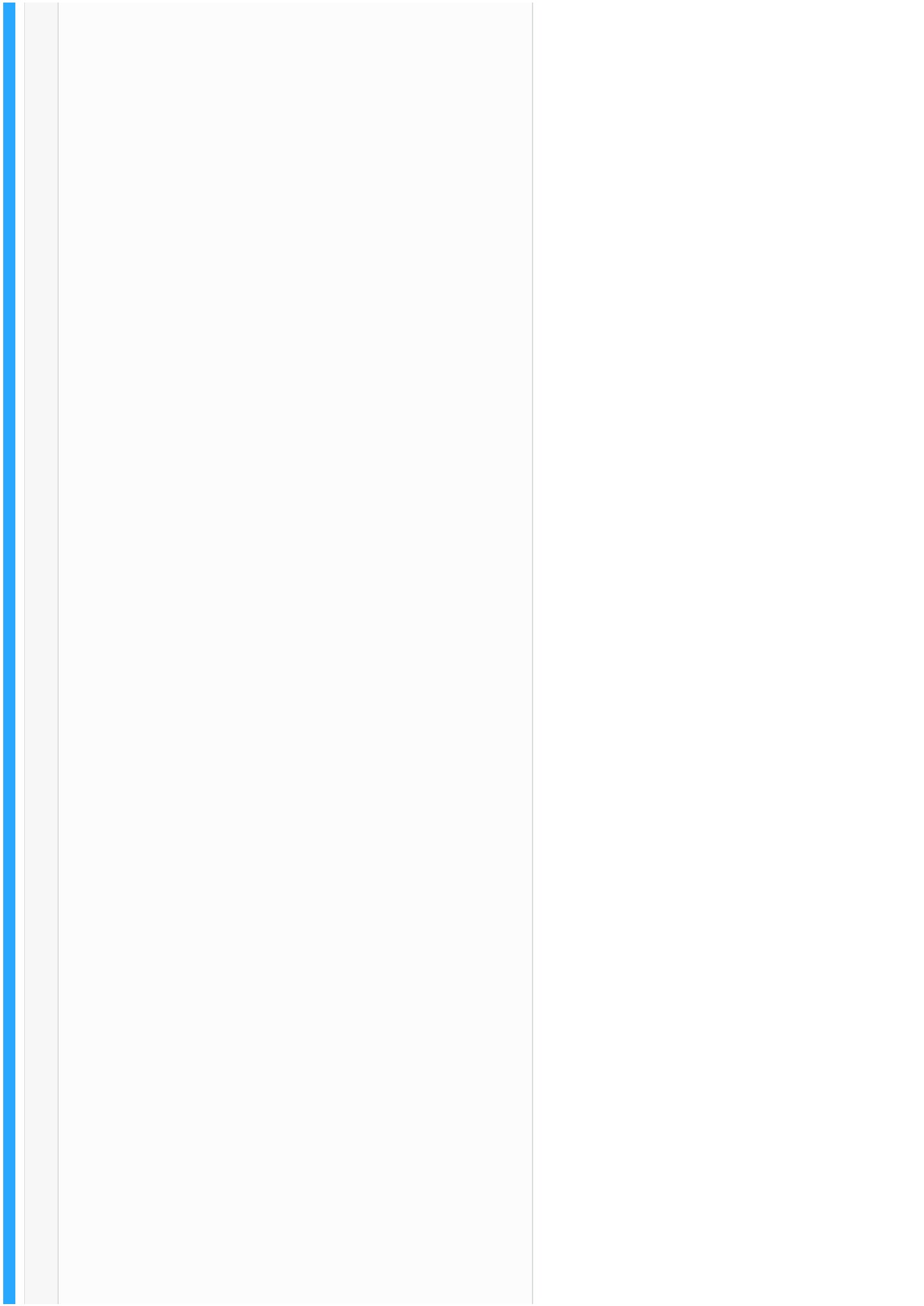


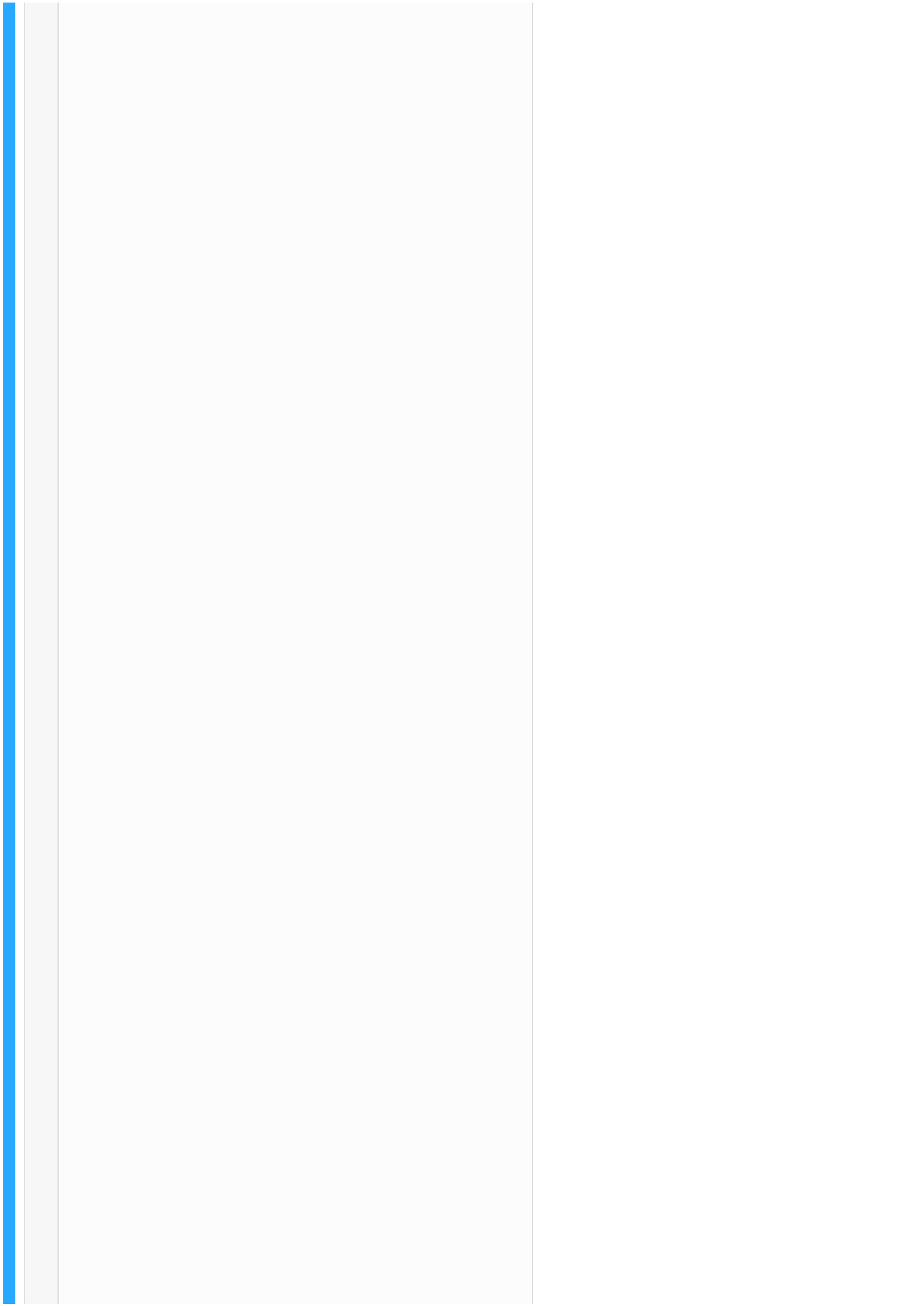


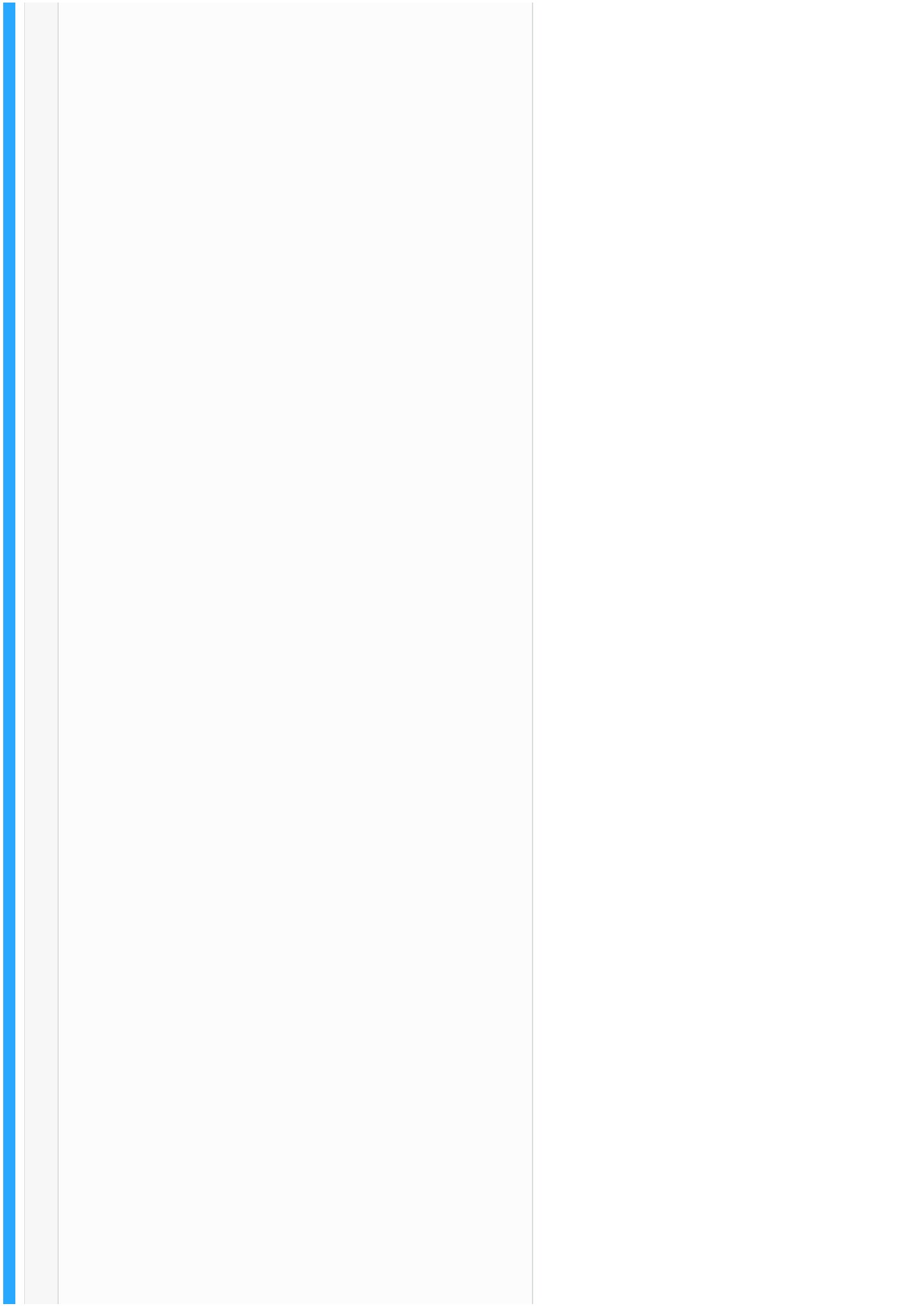
















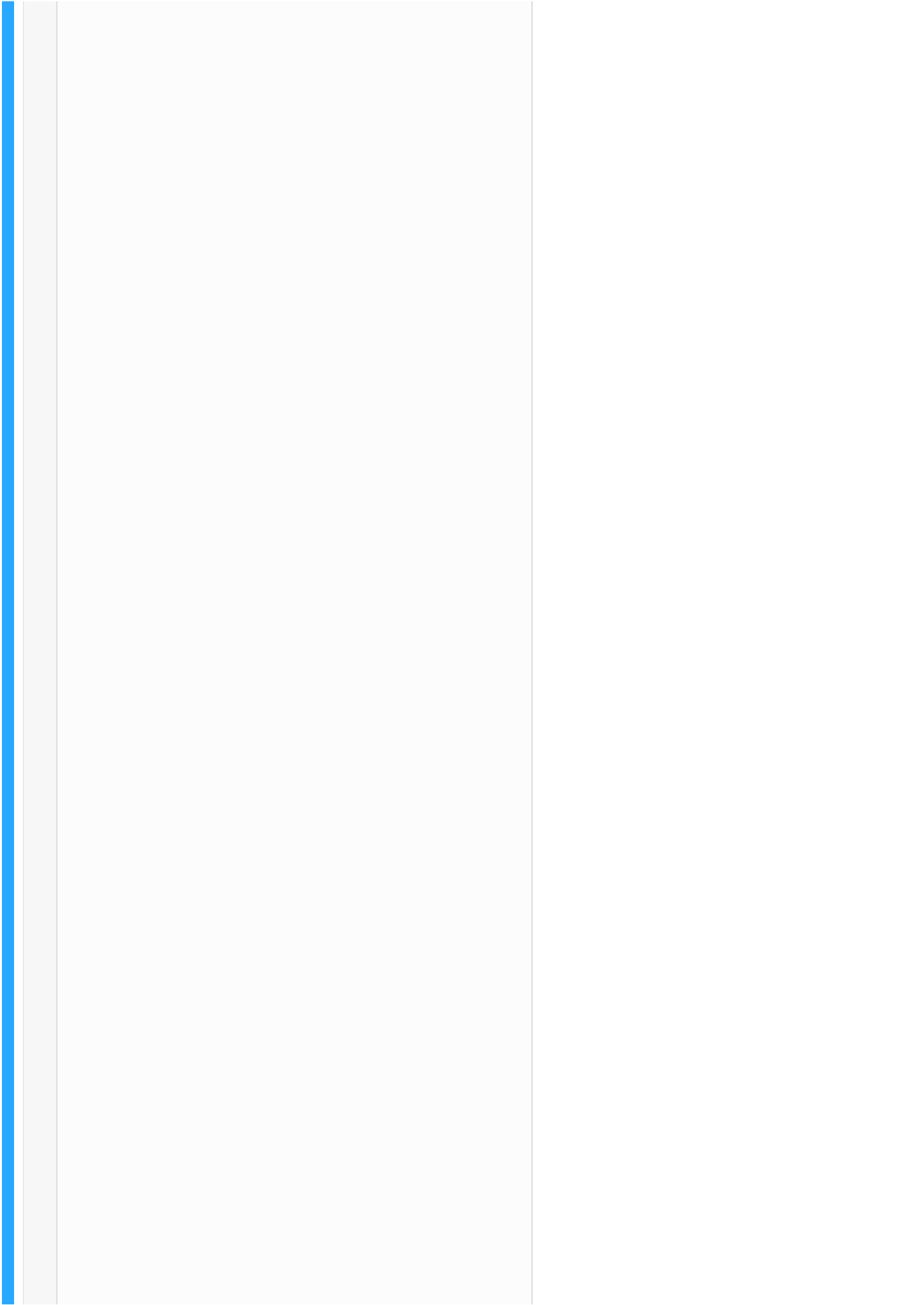
DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)

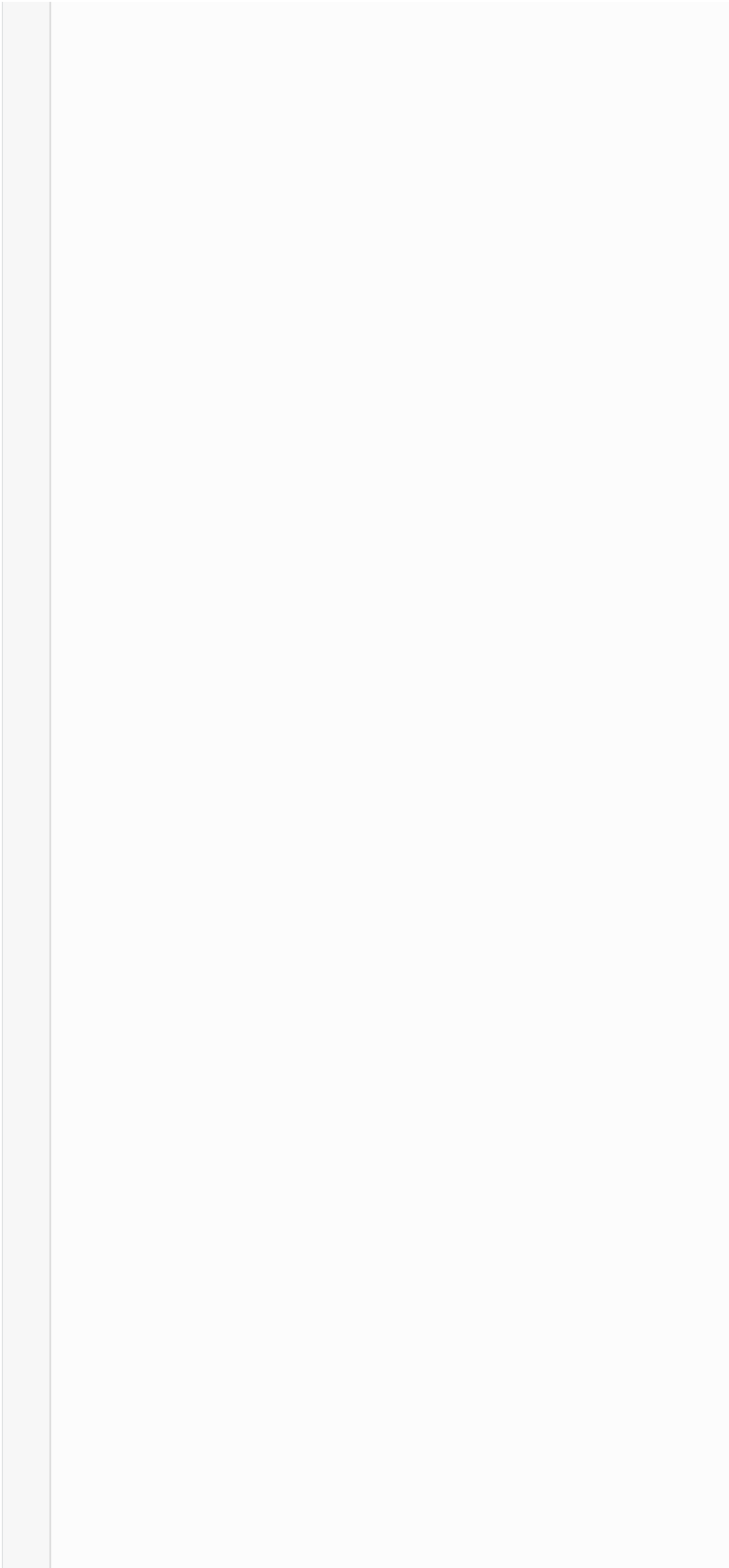


[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)









If you open *http://localhost:4000* in your browser you will access the Cube.js Playground. It is a development environment which generates the Cube.js schema, creates scaffolding for charts, and more. It has its own query builder which lets you generate charts with different charting libraries.

Now, let's move on to building our own query builder.

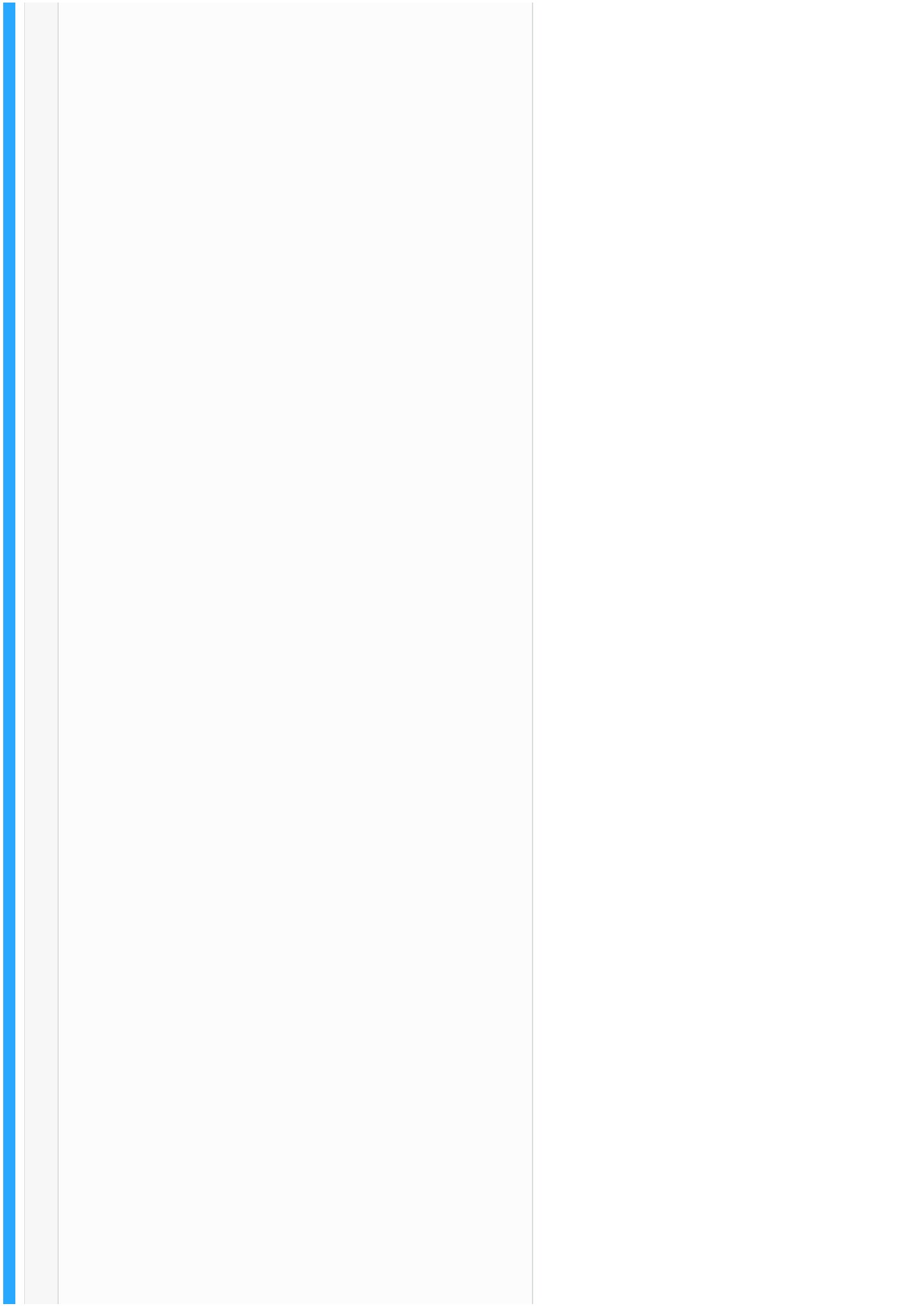
Building a Query Builder

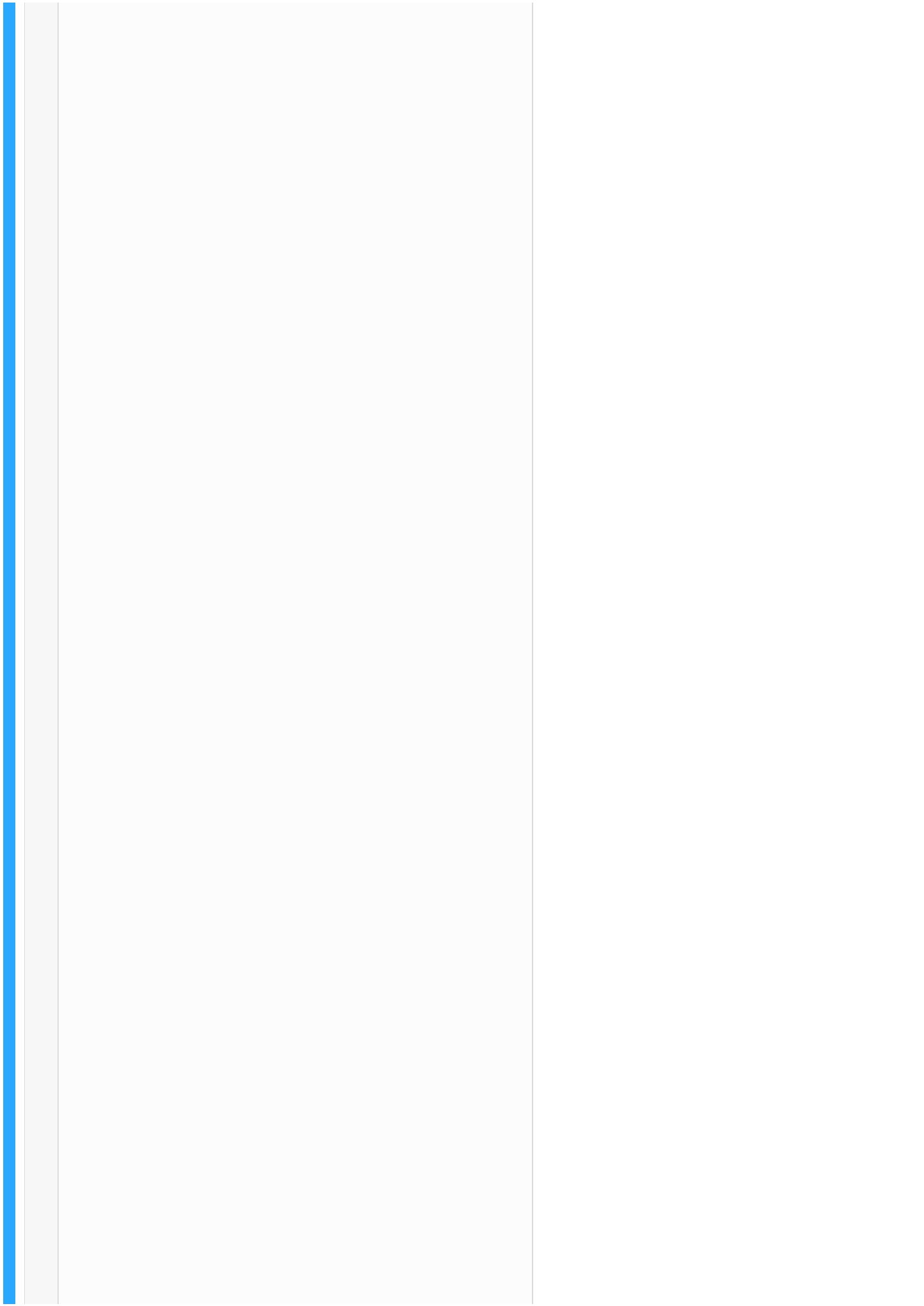
The `<QueryBuilder />` component uses the [render props](#) technique. It acts as a data provider by managing the state and API layer and calls `render` props to let developers implement their render logic.

Besides `render`, the only required prop is `cubejsApi`. It expects an instance of your cube.js API client returned by the `cubejs` method.

[Here you can find a detailed reference of the `<QueryBuilder />` component.](#)

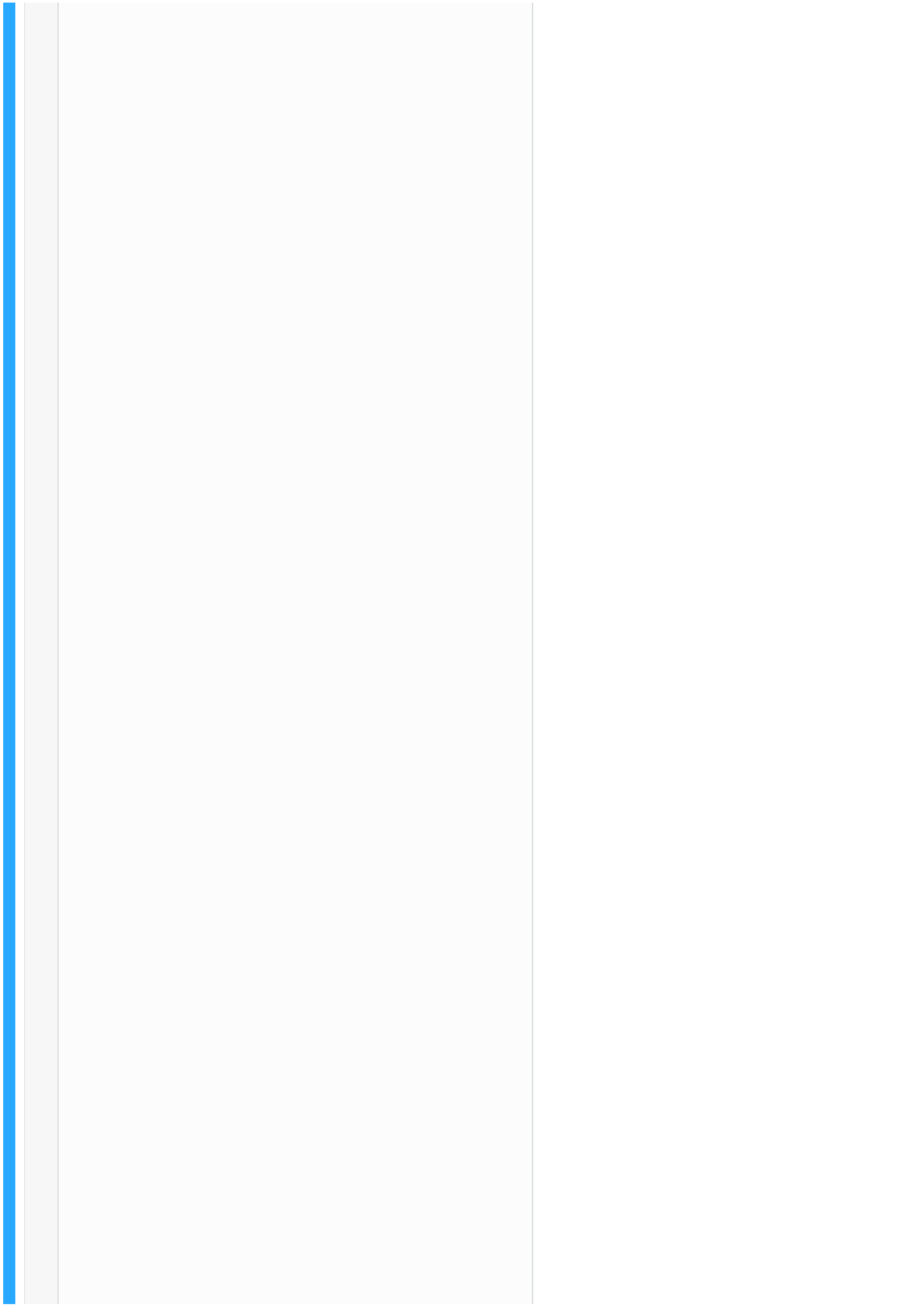
```
1 import cubejs from "@cubejs-client/core";
2 import { QueryBuilder } from "@cubejs-client/react";
3 const cubejsApi = cubejs("CUBEJS_TOKEN", { apiUrl: "CUBEJS_BACKEND_URL" });
4
5 export default () => (
6   <QueryBuilder
7     cubejsApi={cubejsApi}
8     render={queryBuilder => {
9       // Render whatever you want based on the state of queryBuilder
10     }}
11   />
12 );
```

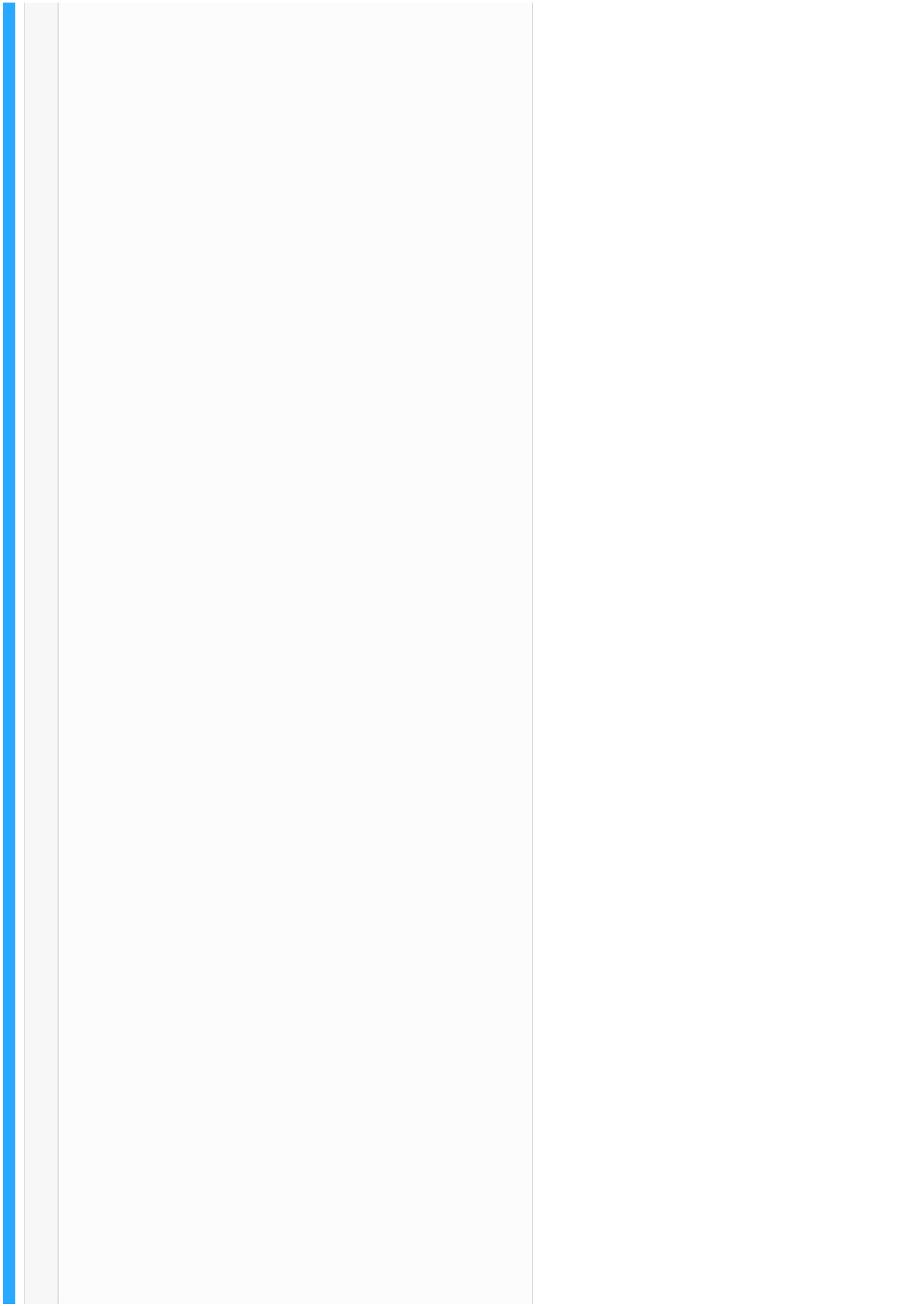




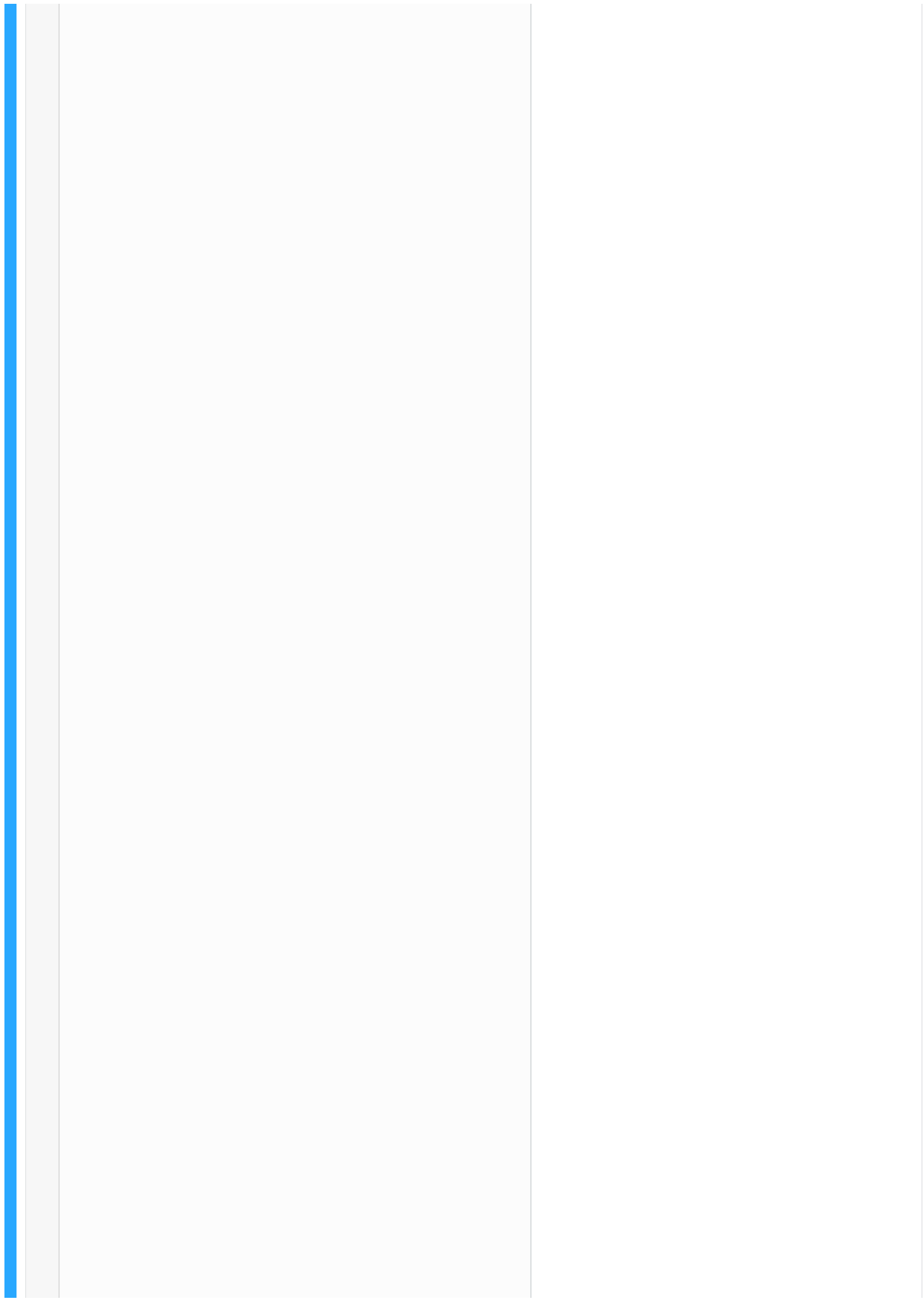








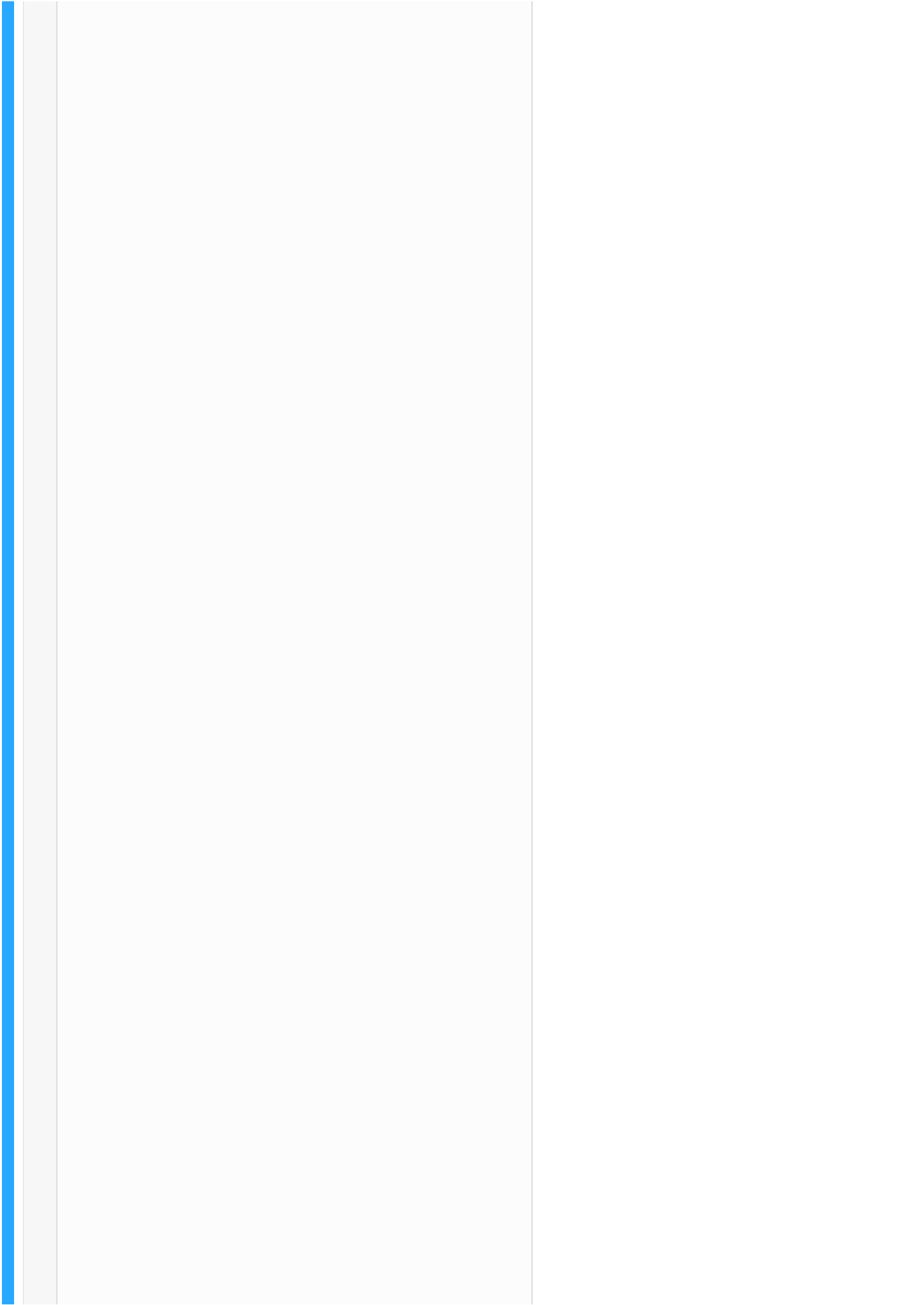










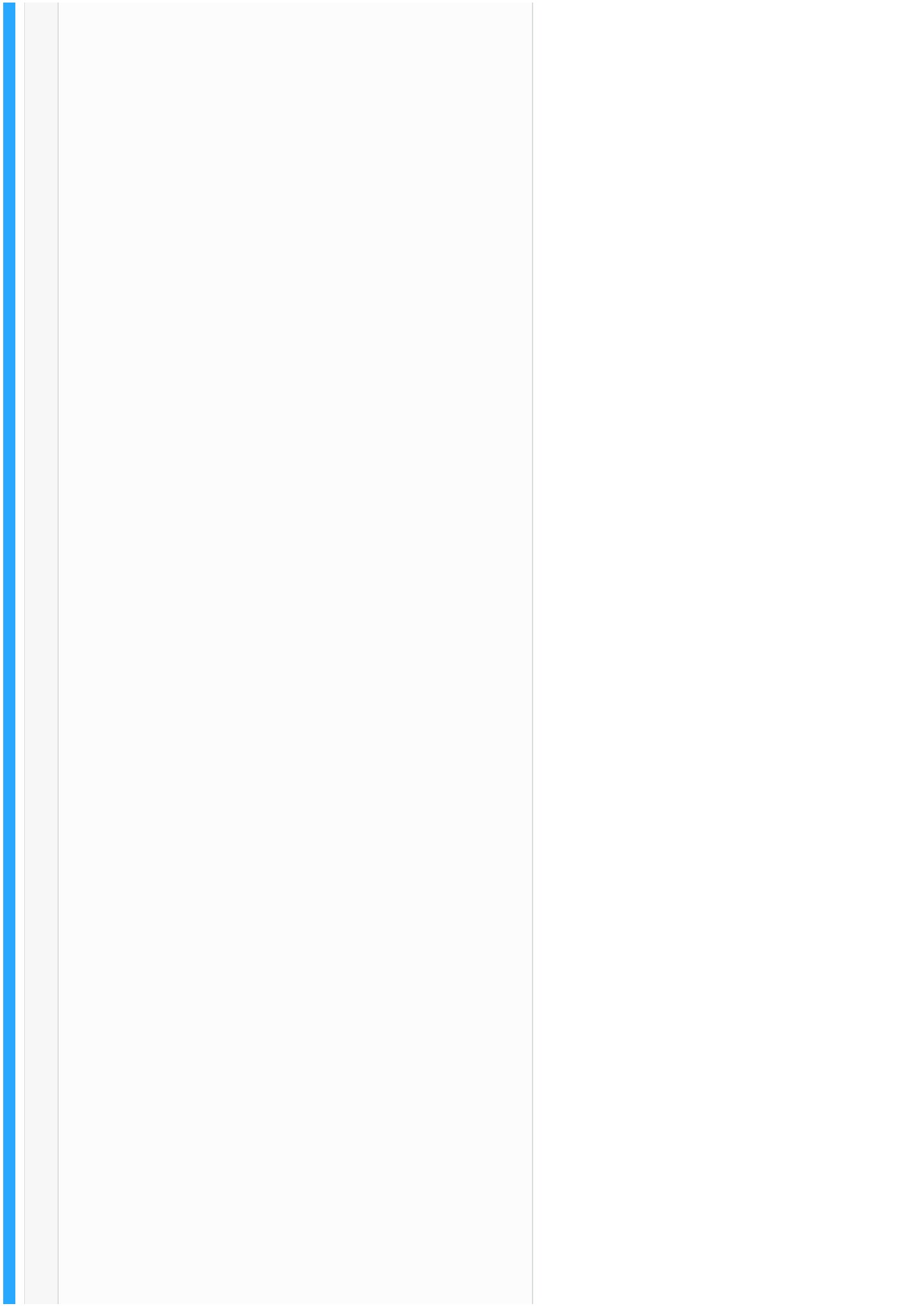
























DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



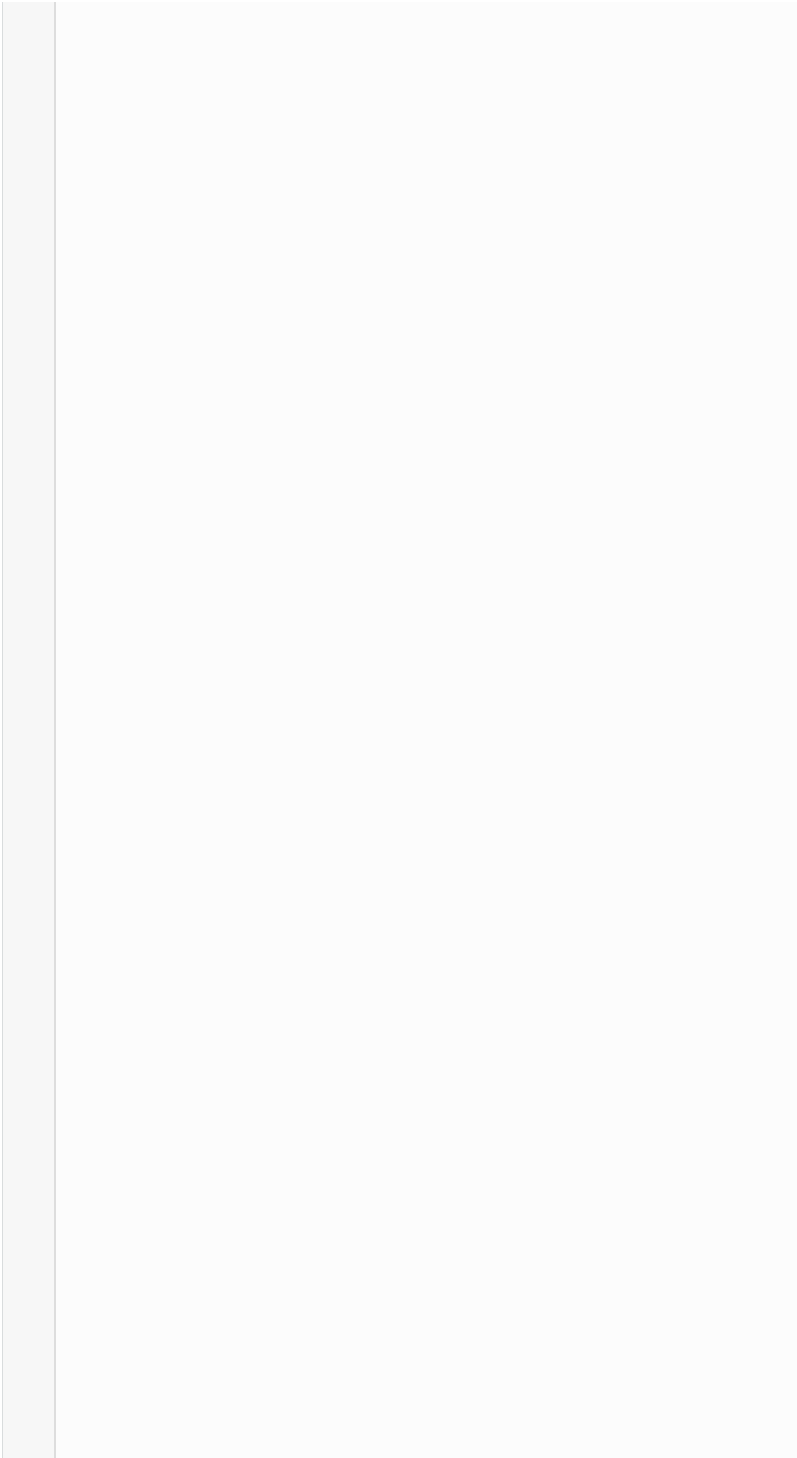
[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)









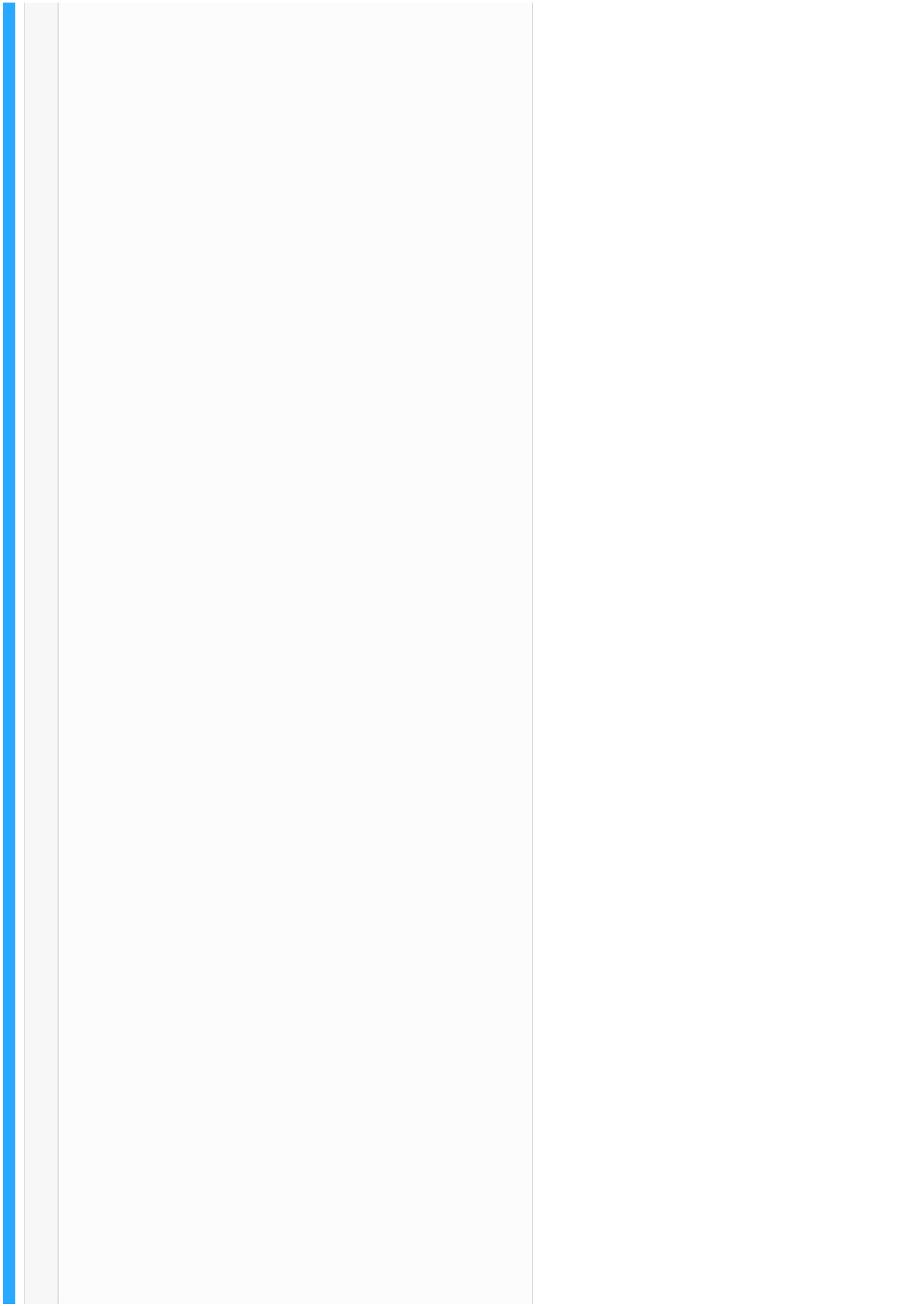


The properties of `queryBuilder` can be split into categories based on what element they are referred to. To render and update measures, you need to use `measures`, `availableMeasures`, and `updateMeasures`.

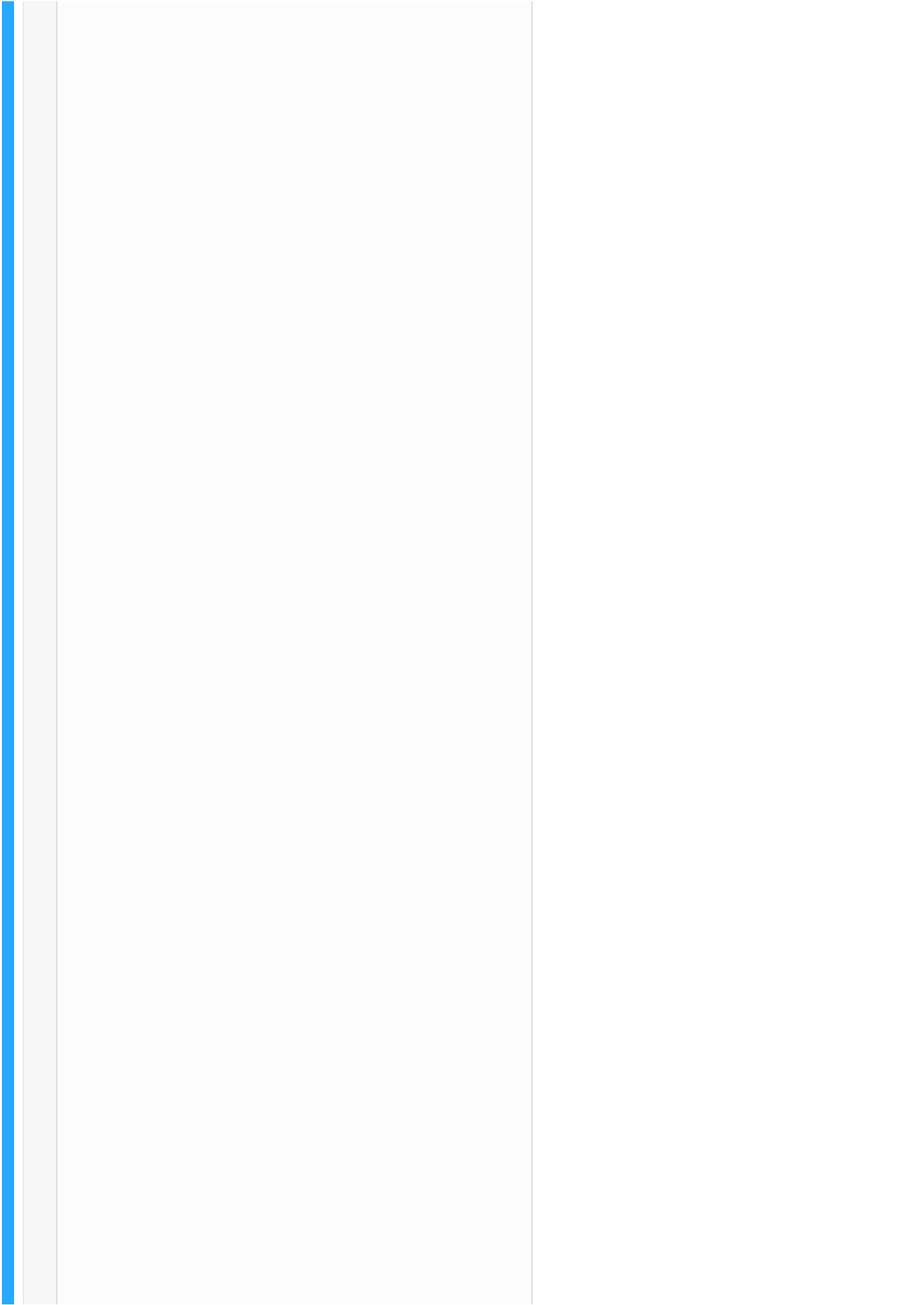
`measures` is an array of already selected measures. It is usually empty in the beginning (unless you passed a default `query` prop). `availableMeasures` is an array of all measures loaded via API from your Cube.js data schema. Both `measures` and `availableMeasures` are arrays of objects with `name`, `title`, `shortTitle`, and `type` keys. `name` is used as an ID. `title` could be used

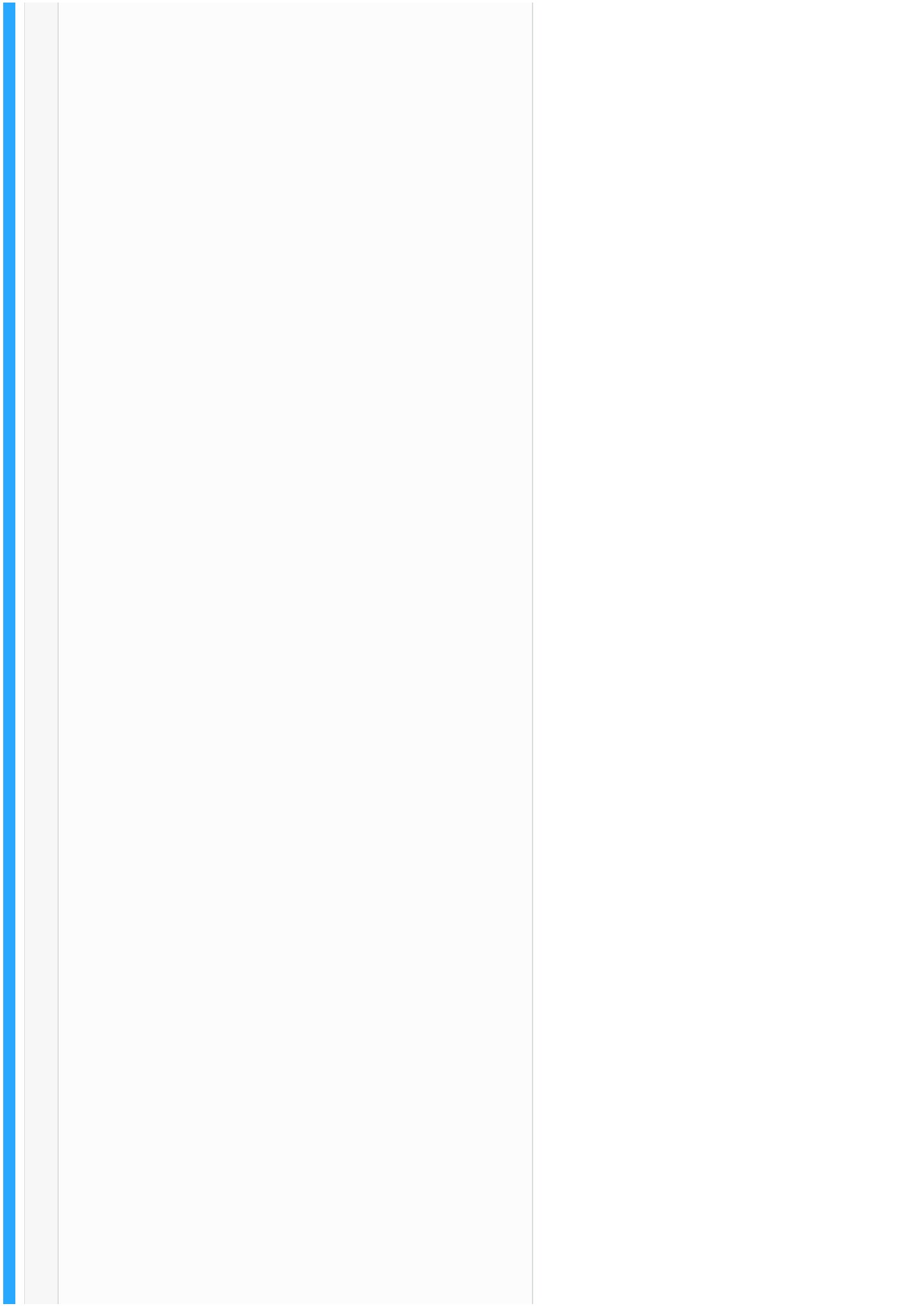
as a human-readable name, and `shortTitle` is only the measure's title without the Cube's title.

```
1 // `measures` and `availableMeasures` are arrays
  with the following structure
2 [
3 { name: "Orders.count", title: "Orders Count", s
  hortTitle: "Count", type: "number" },
4 { name: "Orders.number", title: "Orders Number",
  shortTitle: "Number", type: "number" }
5 ]
```



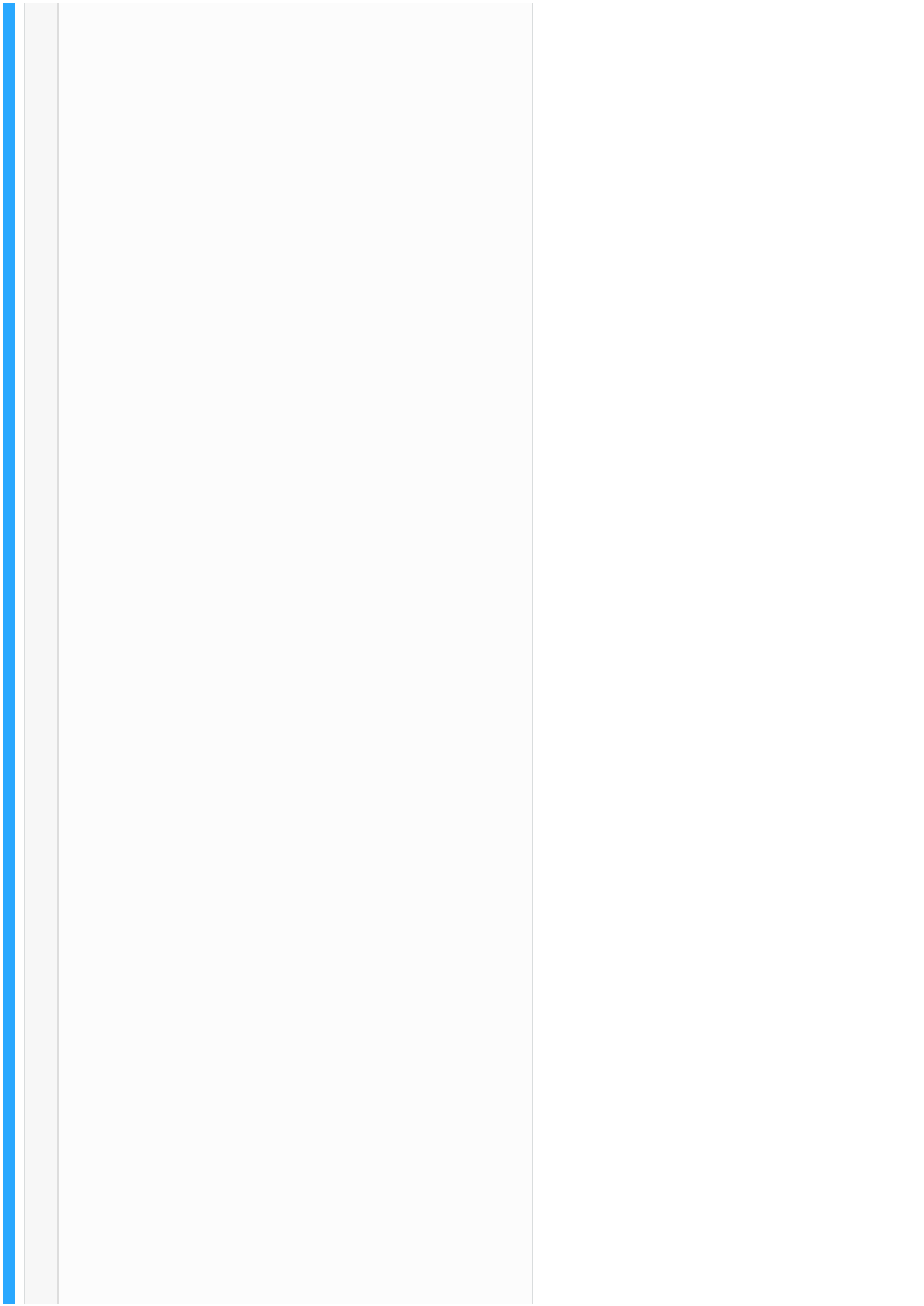




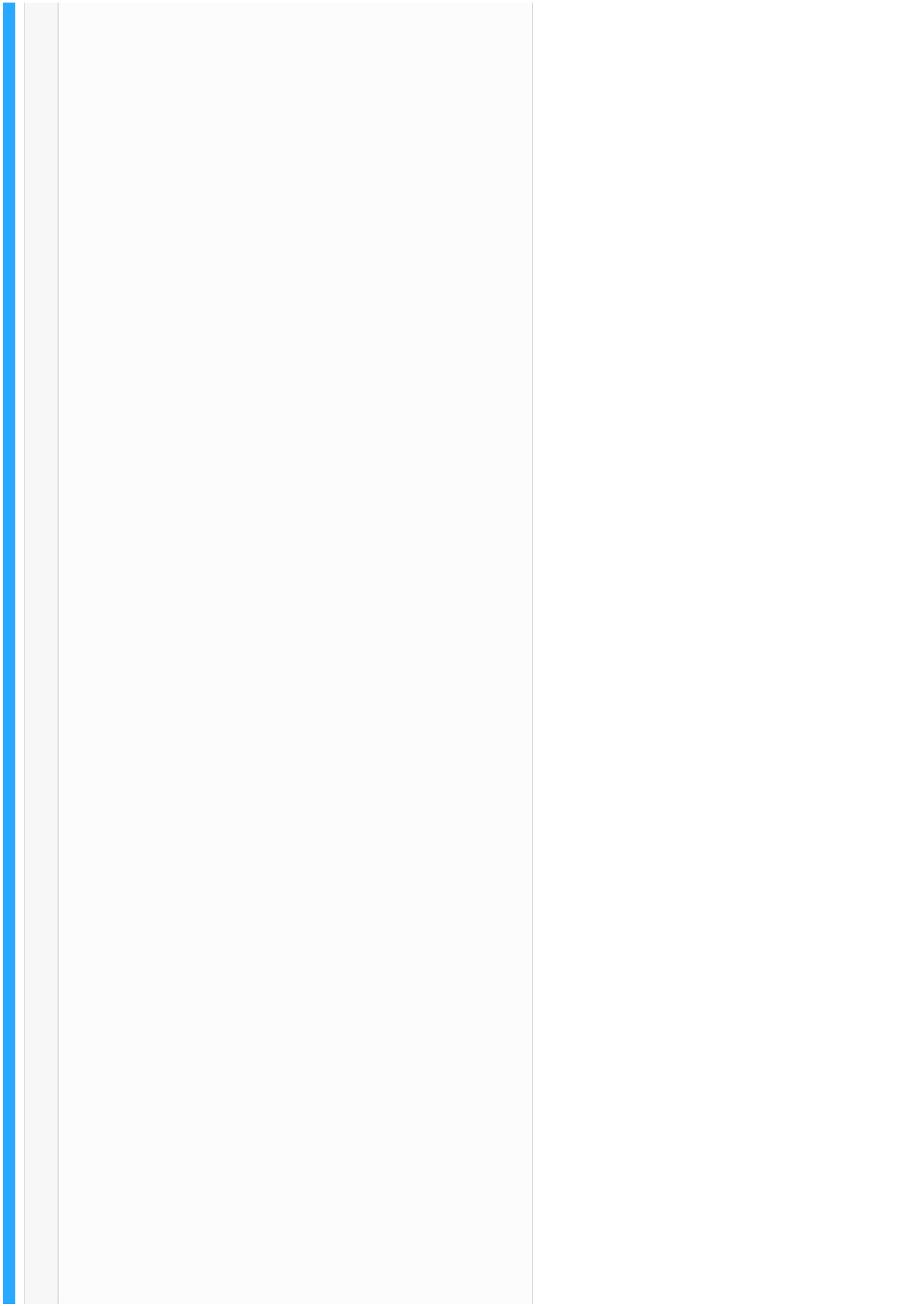






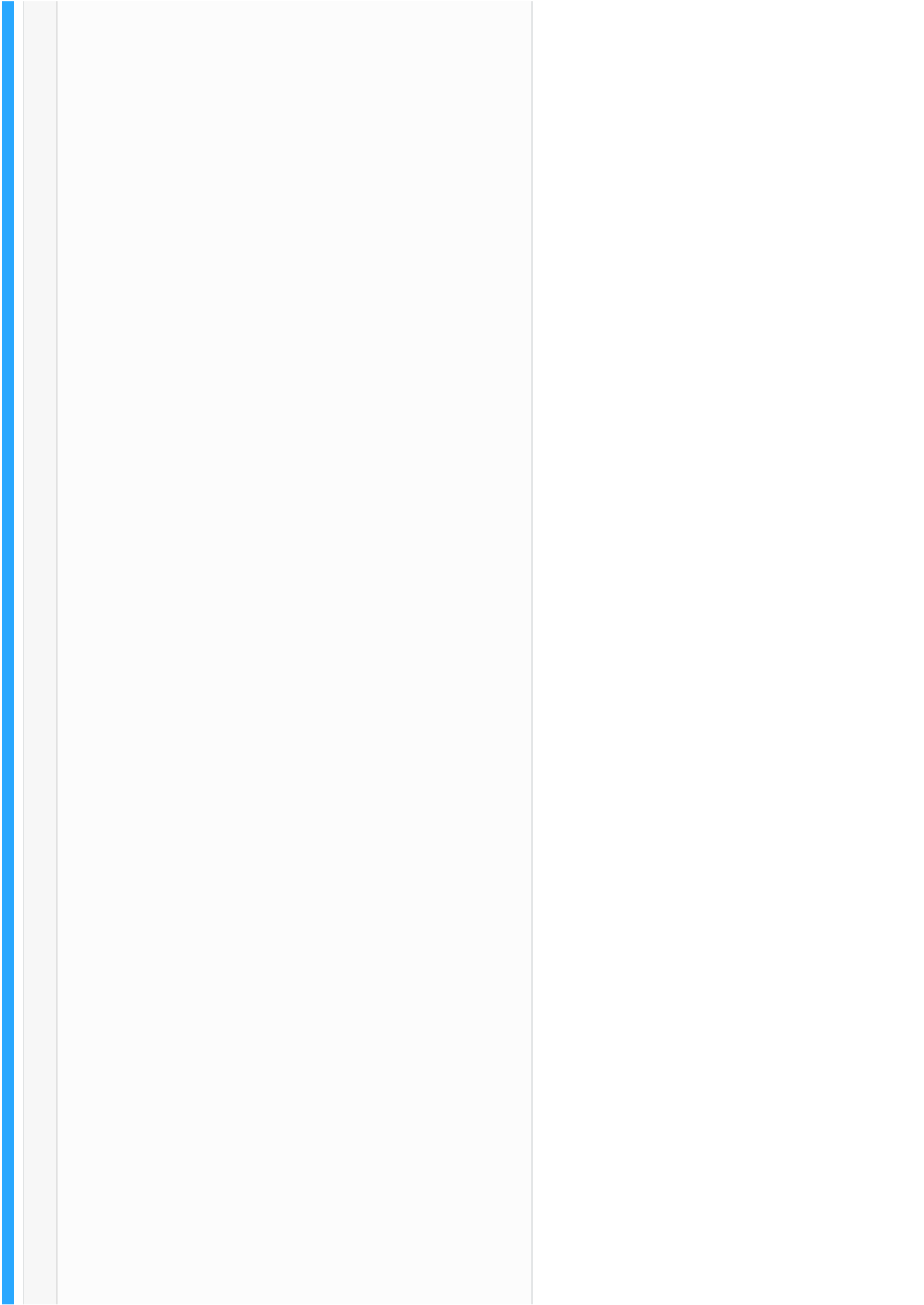


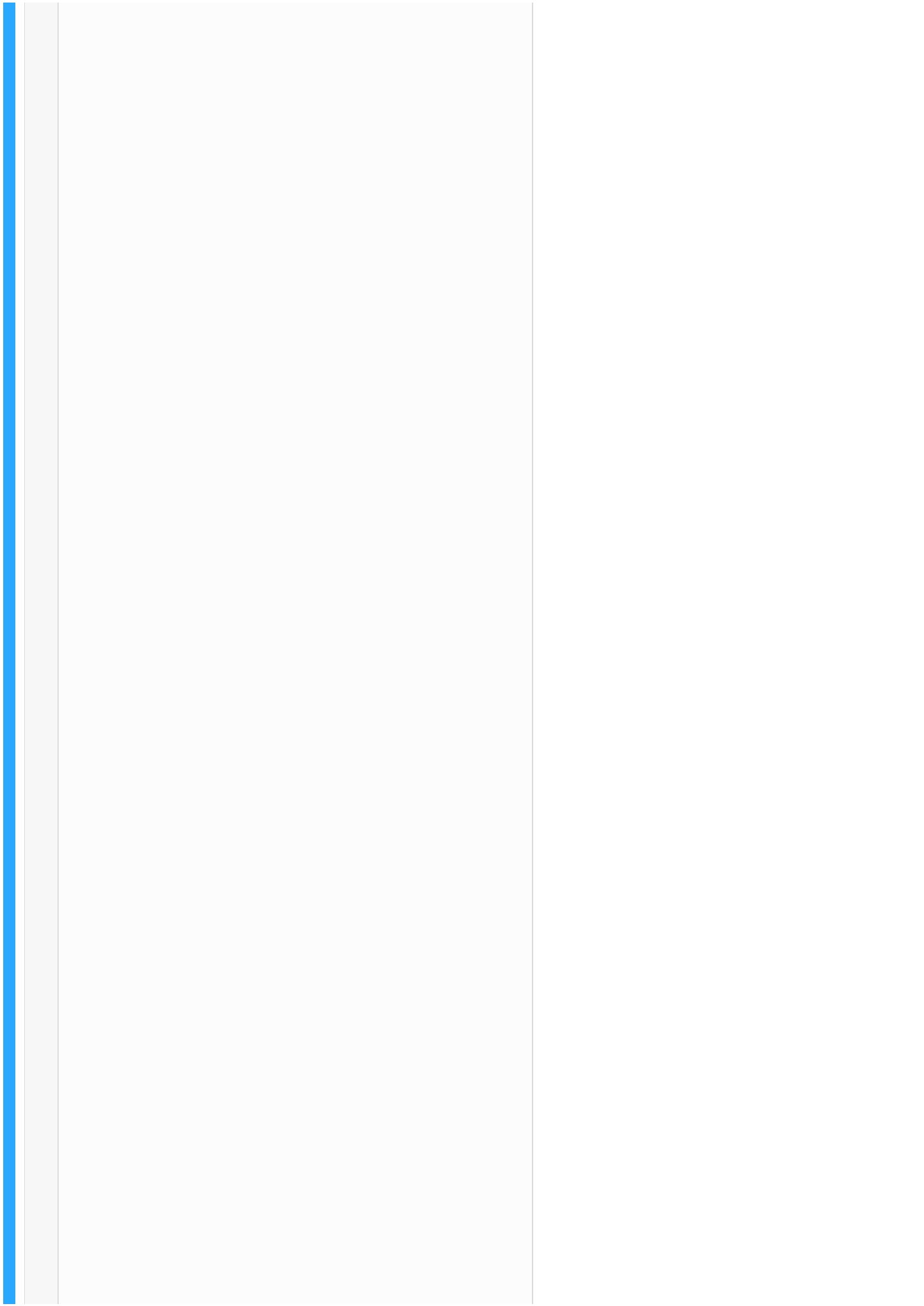






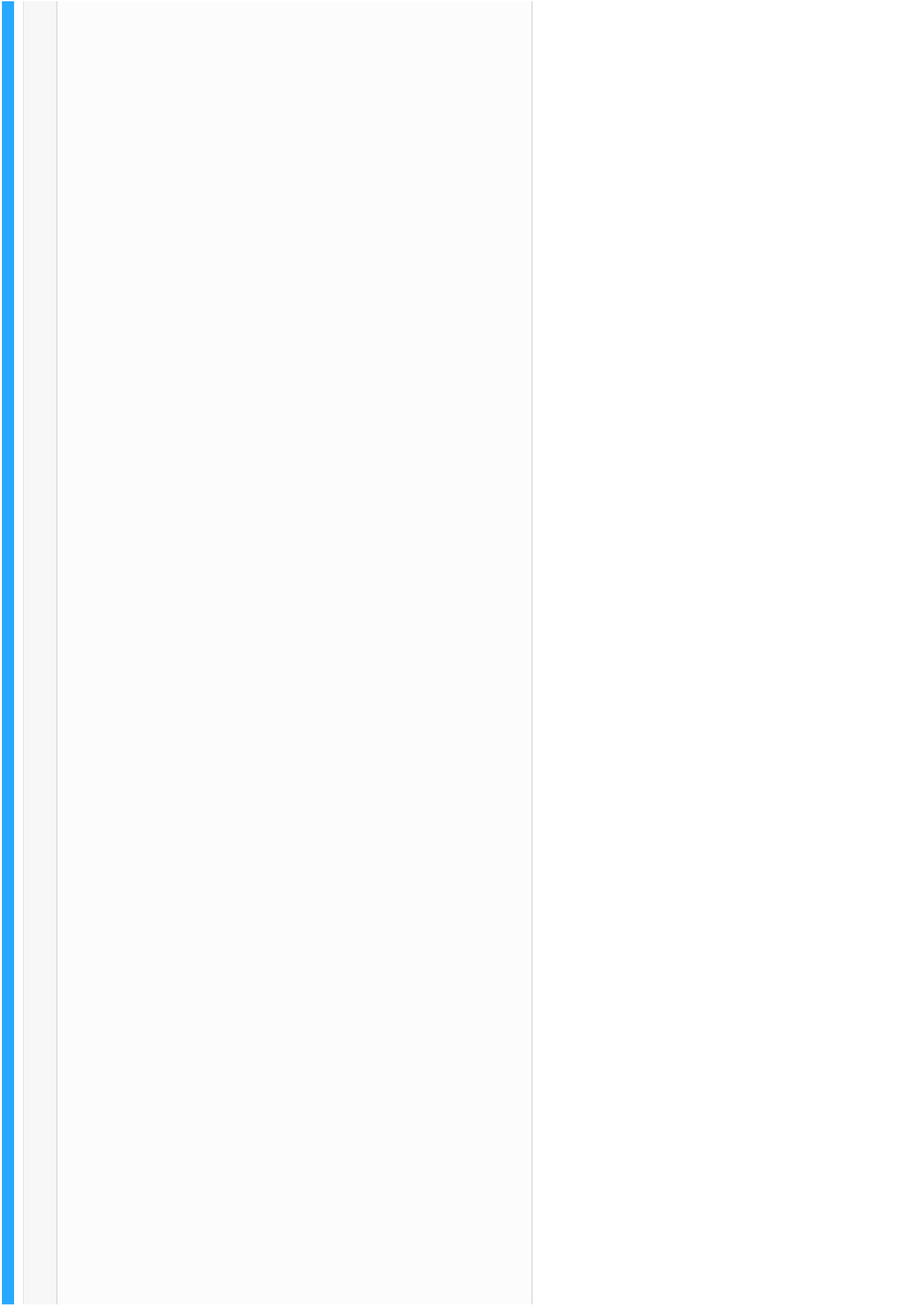


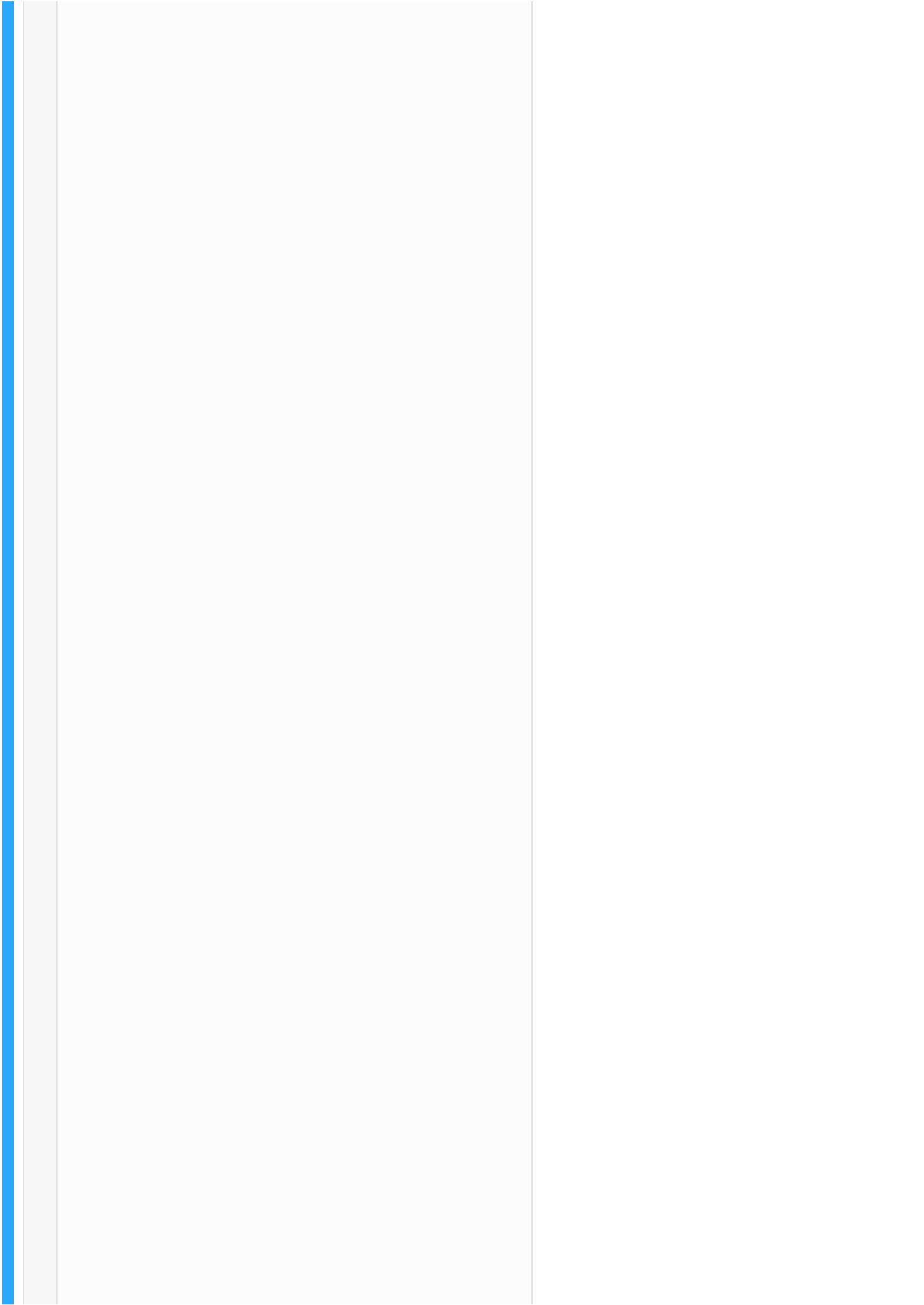






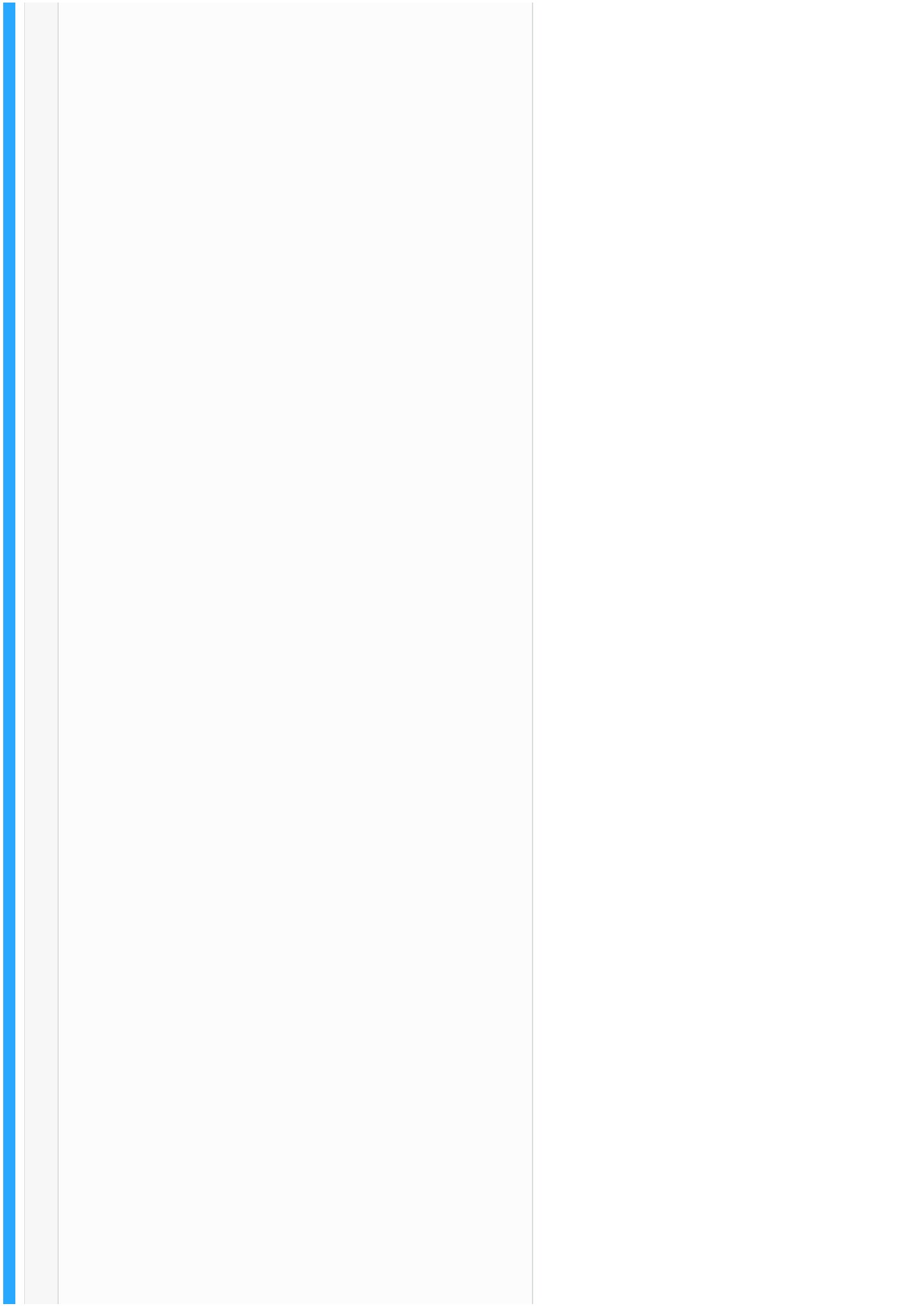
















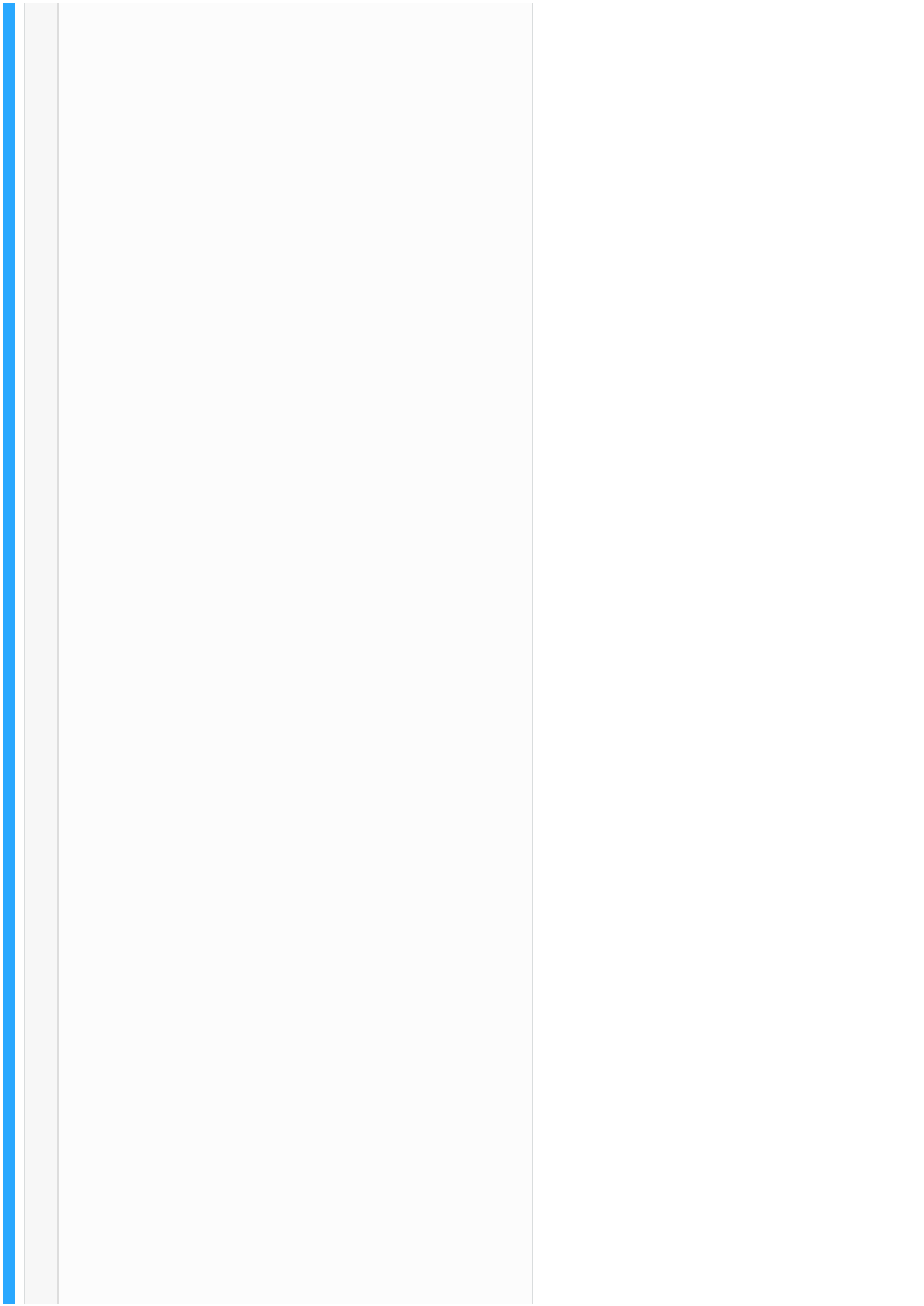
DZone

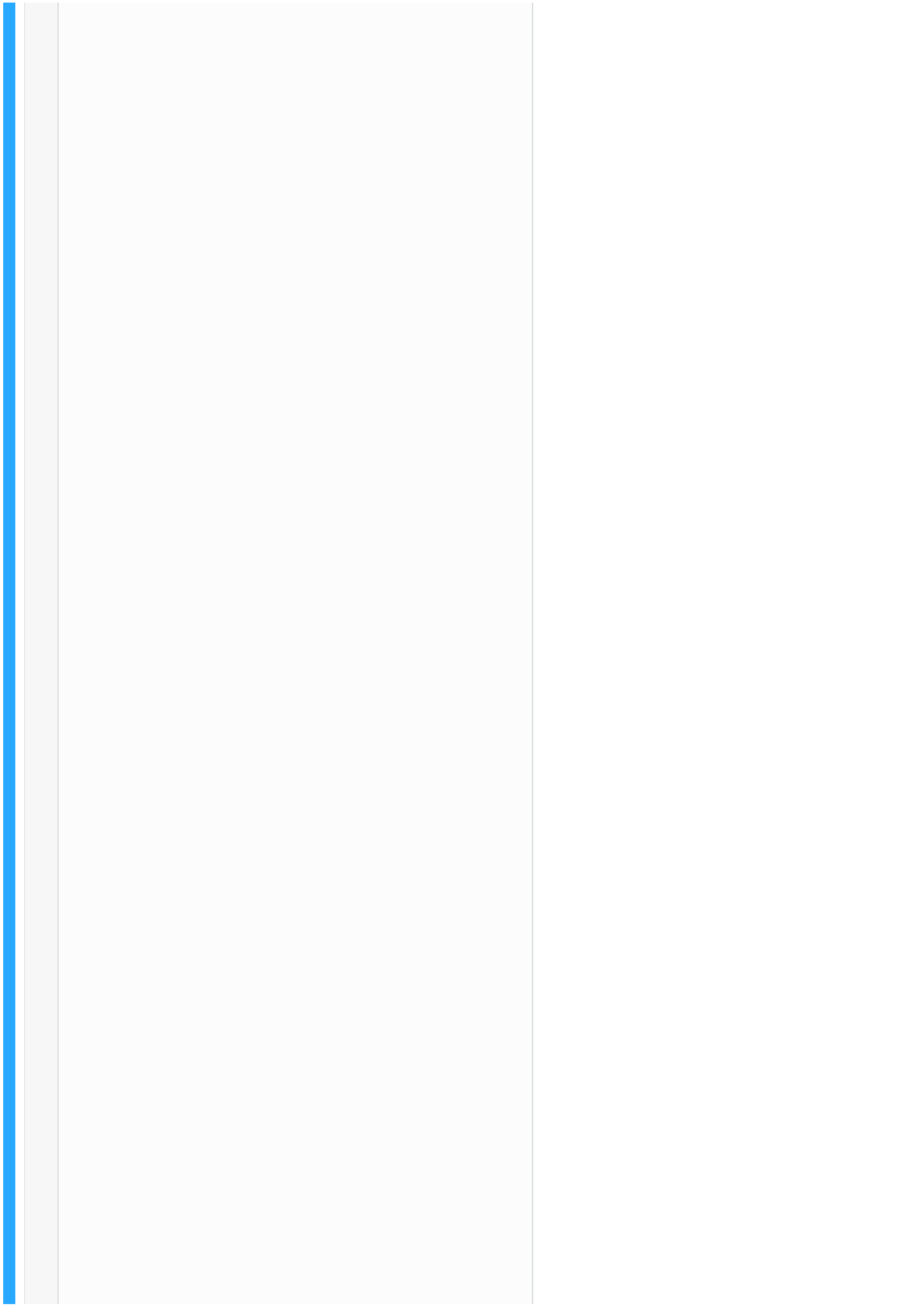
Web Dev Zone

[Log In](#) / [Sign Up](#)

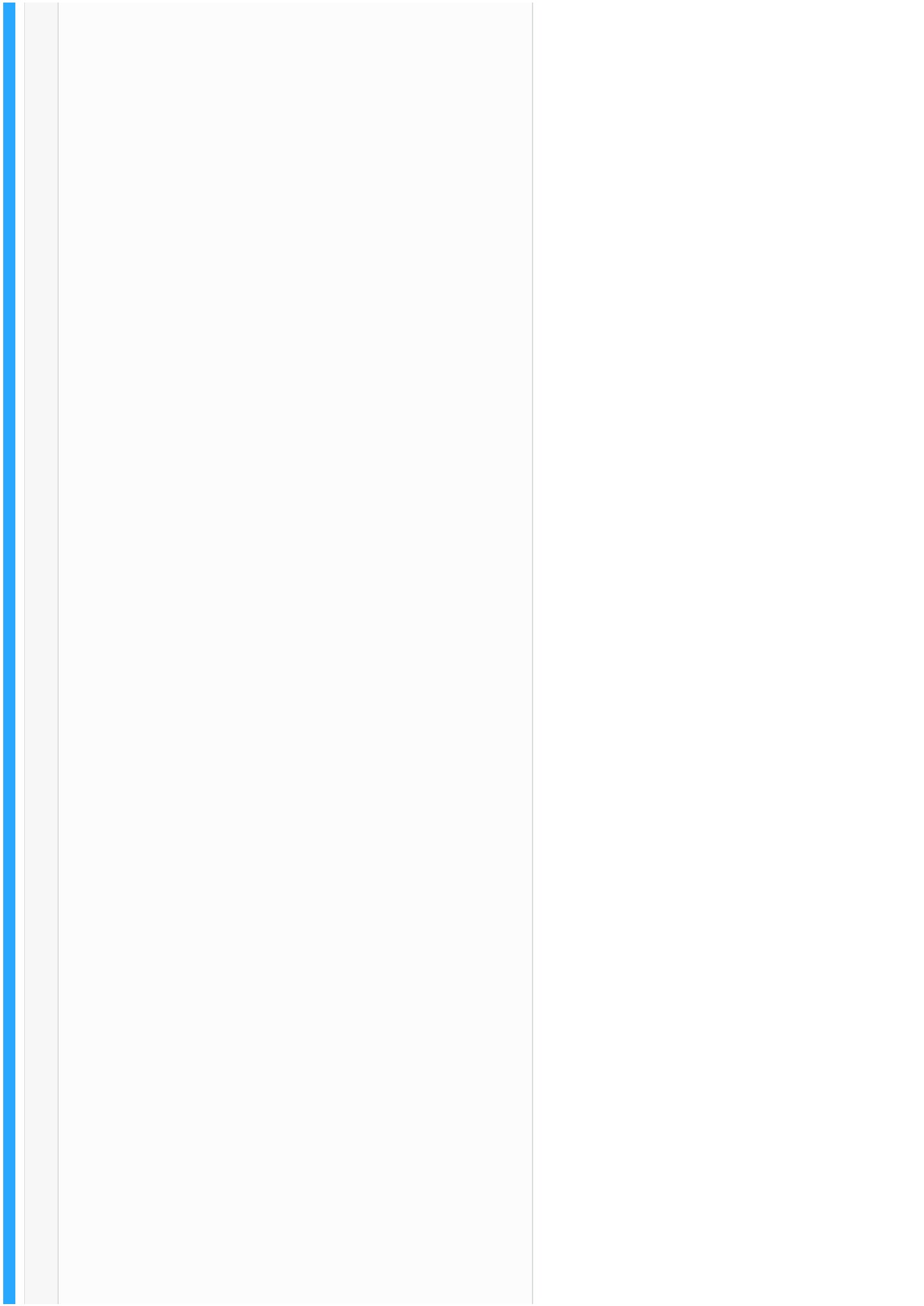


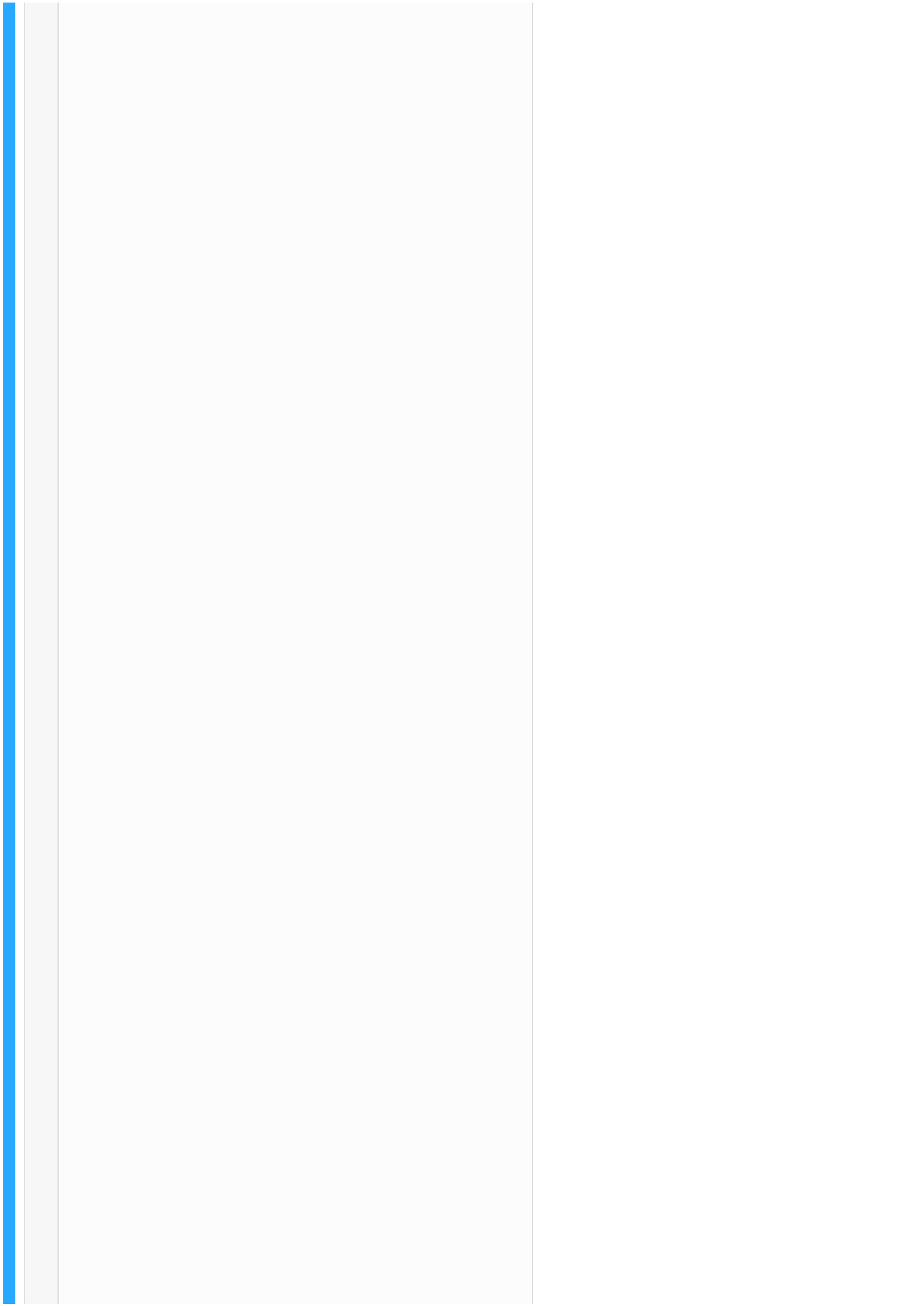
[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)











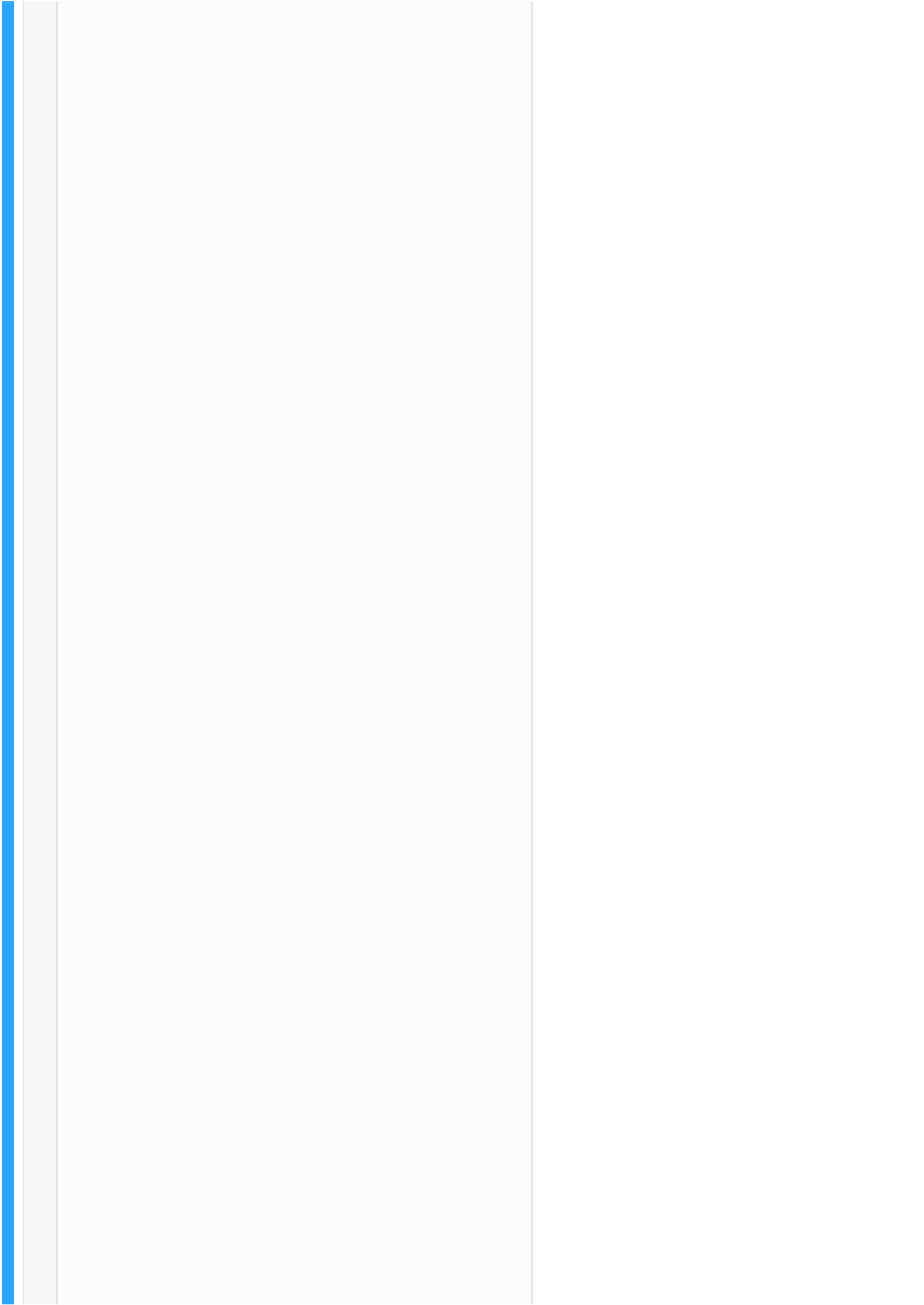
`updateMeasures` is an object with three functions: `add`, `remove`, and `update`. It is used to control the state of the query builder related to measures.

Now, using these properties, we can render a UI to manage measures and render a simple line chart, which will dynamically change the content based on the state of the query builder.

```
1 import React from "react";
2 import ReactDOM from "react-dom";
3 import { Layout, Divider, Empty, Select } from "
  antd";
4 import { QueryBuilder } from "@cubejs-client/rea
  ct";
5 import cubejs from "@cubejs-client/core";
6 import "antd/dist/antd.css";
7
8 import ChartRenderer from "../ChartRenderer";
9
10 const cubejsApi = cubejs(
11   "YOUR-CUBEJS-API-TOKEN",
12   { apiUrl: "http://localhost:4000/cubejs-api/v1"
13   }
14 );
15
16 const App = () => (
17   <QueryBuilder
18     query={{
19       timeDimensions: [
```

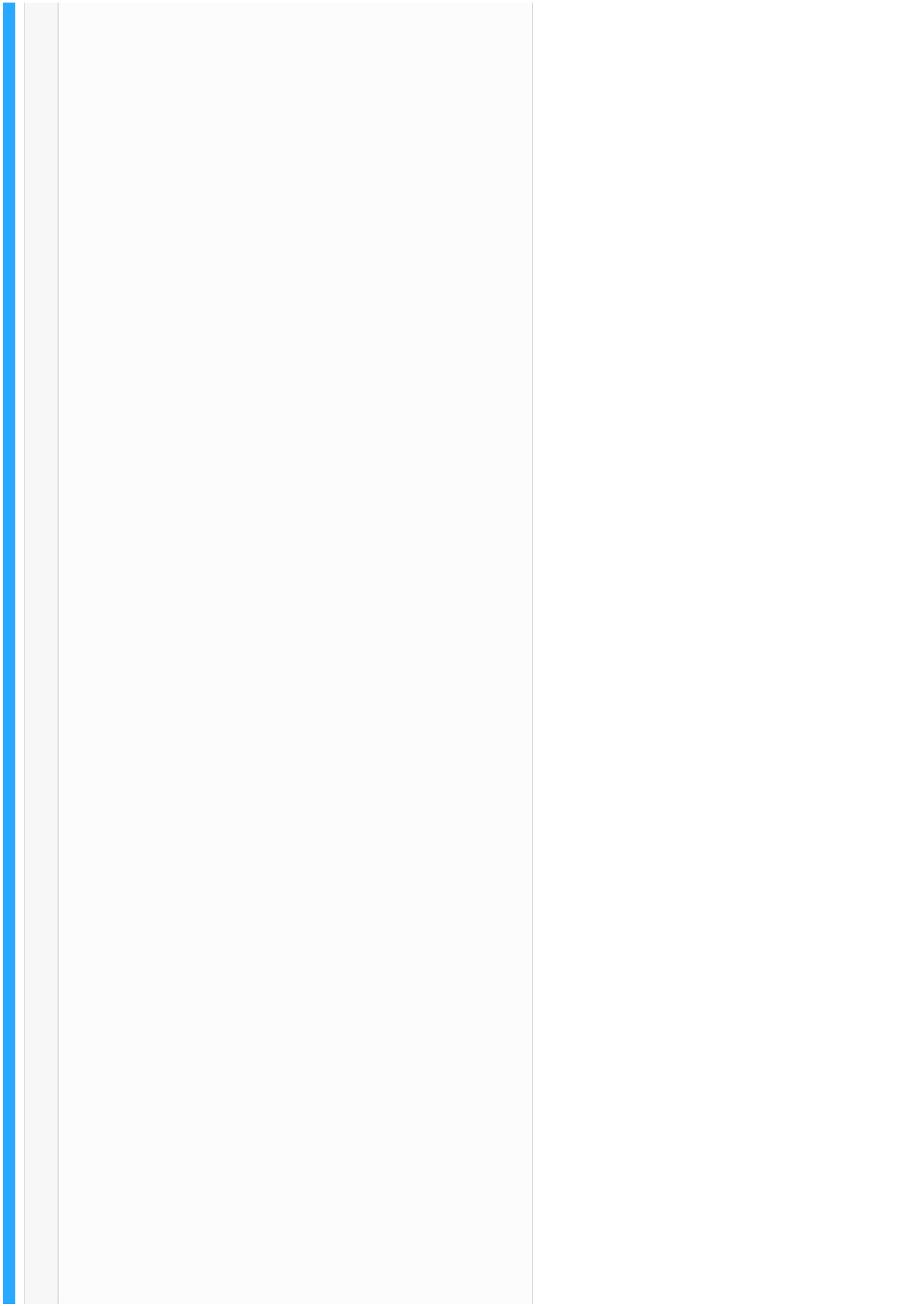


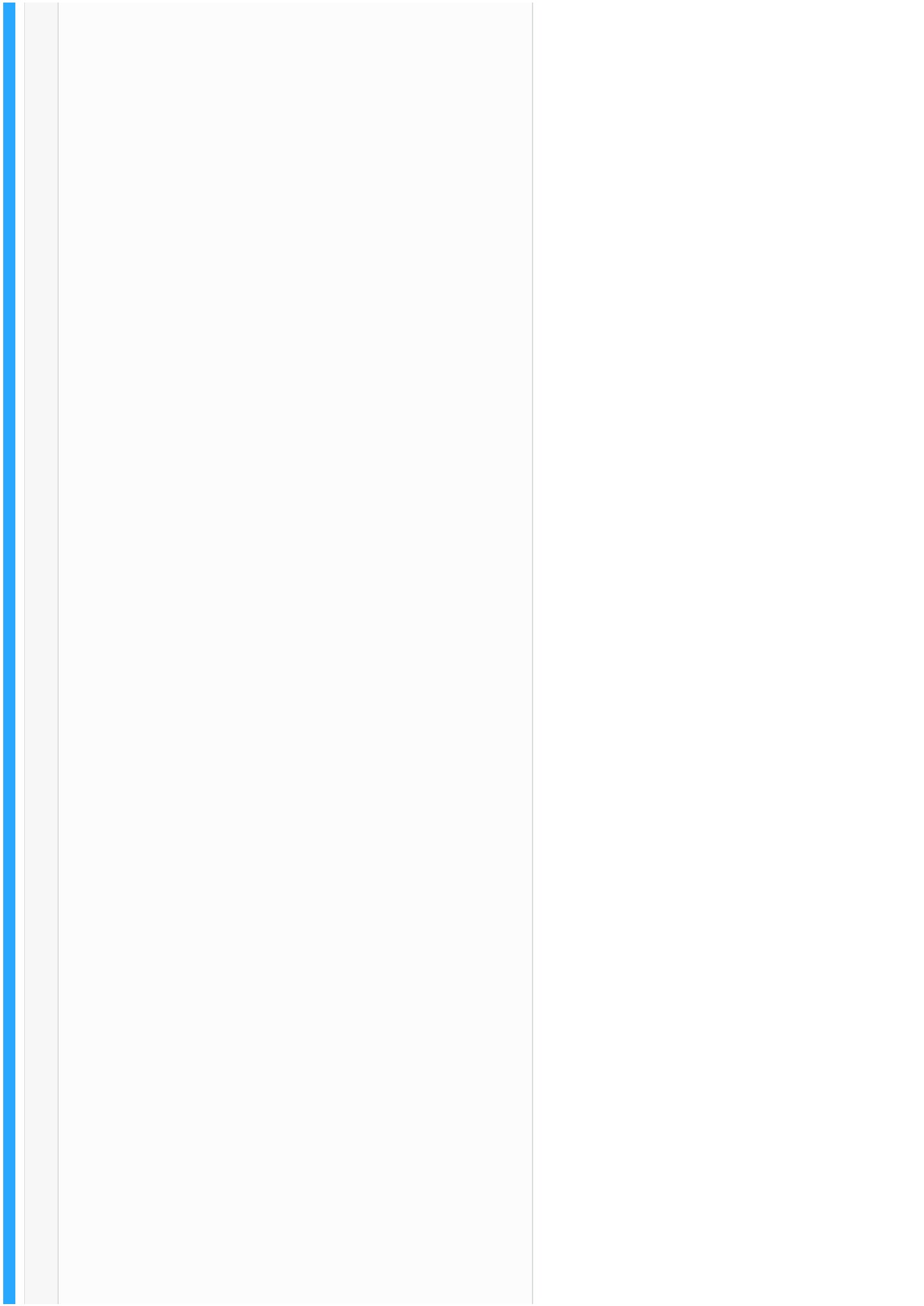
```
20 dimension: "LineItems.createdAt",
21 granularity: "month"
22 }
23 ]
24 }}
25 cubejsApi={cubejsApi}
26 render={({ resultSet, measures, availableMeasure
s, updateMeasures }) => (
27 <Layout.Content style={{ padding: "20px" }}>
28 <Select
29 mode="multiple"
30 style={{ width: "100%" }}
31 placeholder="Please select"
32 onSelect={measure => updateMeasures.add(measure)
}
33 onDeselect={measure => updateMeasures.remove(mea
sure)}
34 >
35 {availableMeasures.map(measure => (
36 <Select.Option key={measure.name} value={measure
}>
37 {measure.title}
38 </Select.Option>
39 )))
40 </Select>
41 <Divider />
42 {measures.length > 0 ? (
43 <ChartRenderer resultSet={resultSet} />
44 ) : (
45 <Empty description="Select measure or dimension
to get started" />
46 )}
47 </Layout.Content>
48 )}
49 />
50 );
51
52 const rootElement = document.getElementById("roo
t");
53 ReactDOM.render(<App />, rootElement);
```

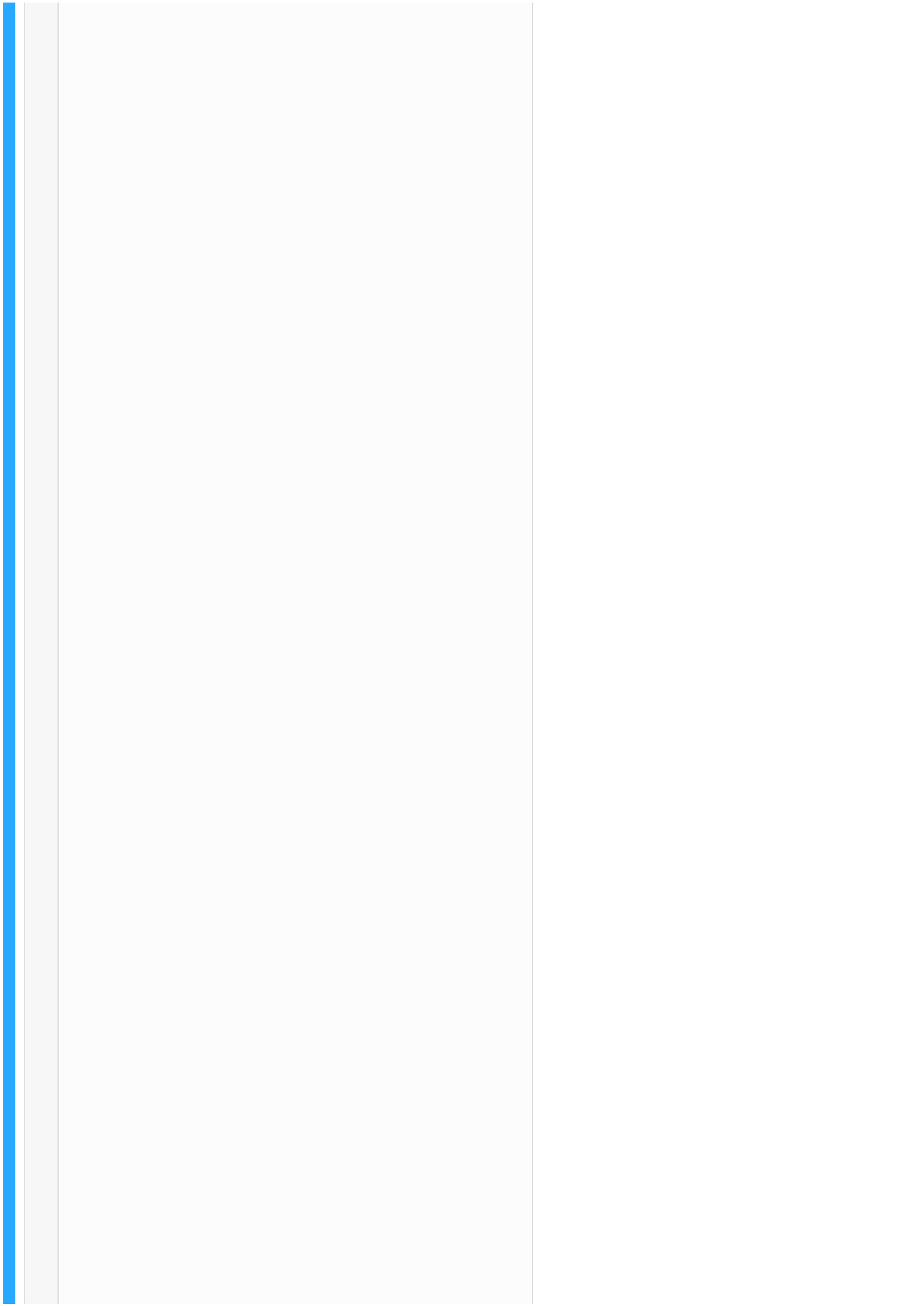








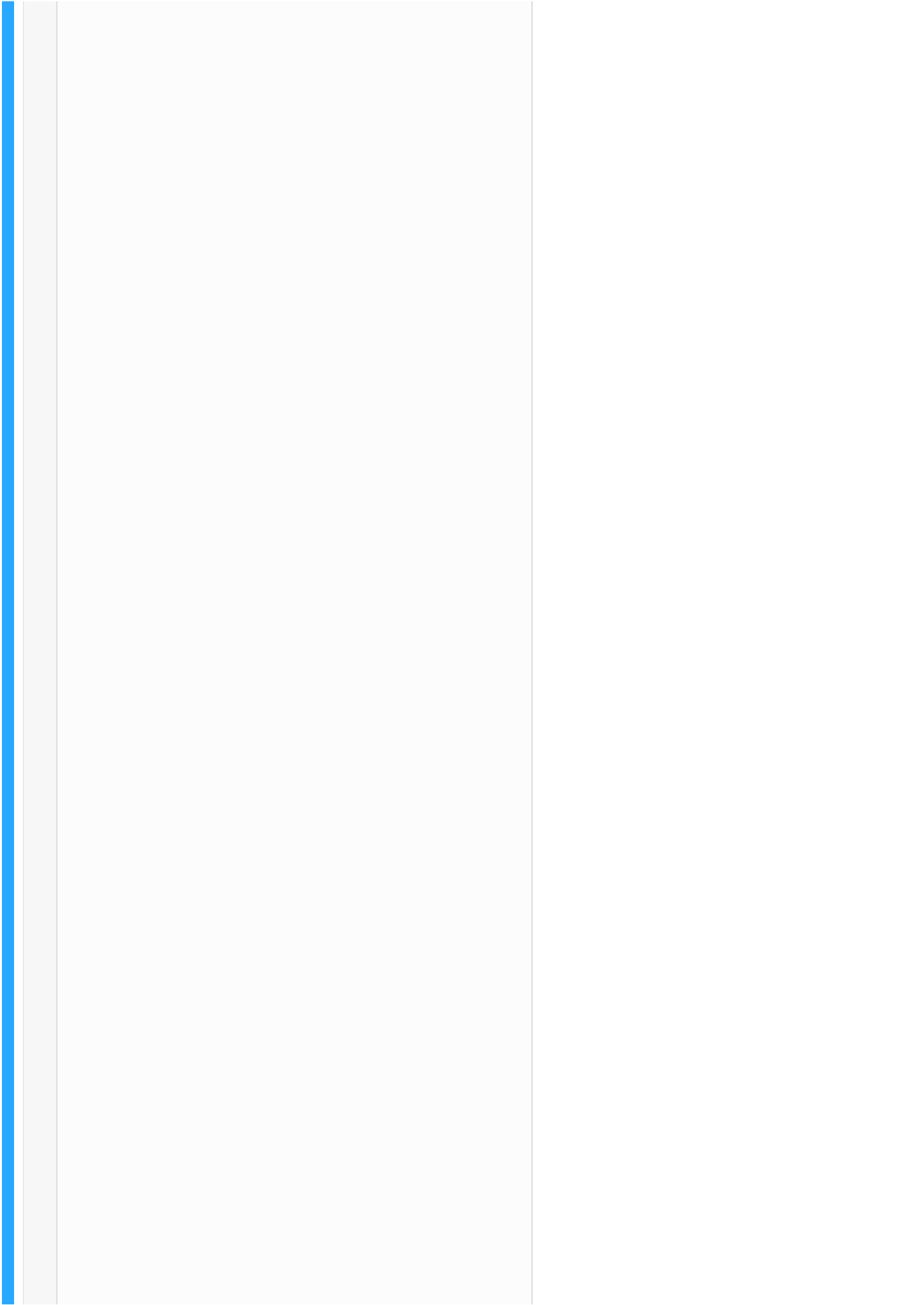






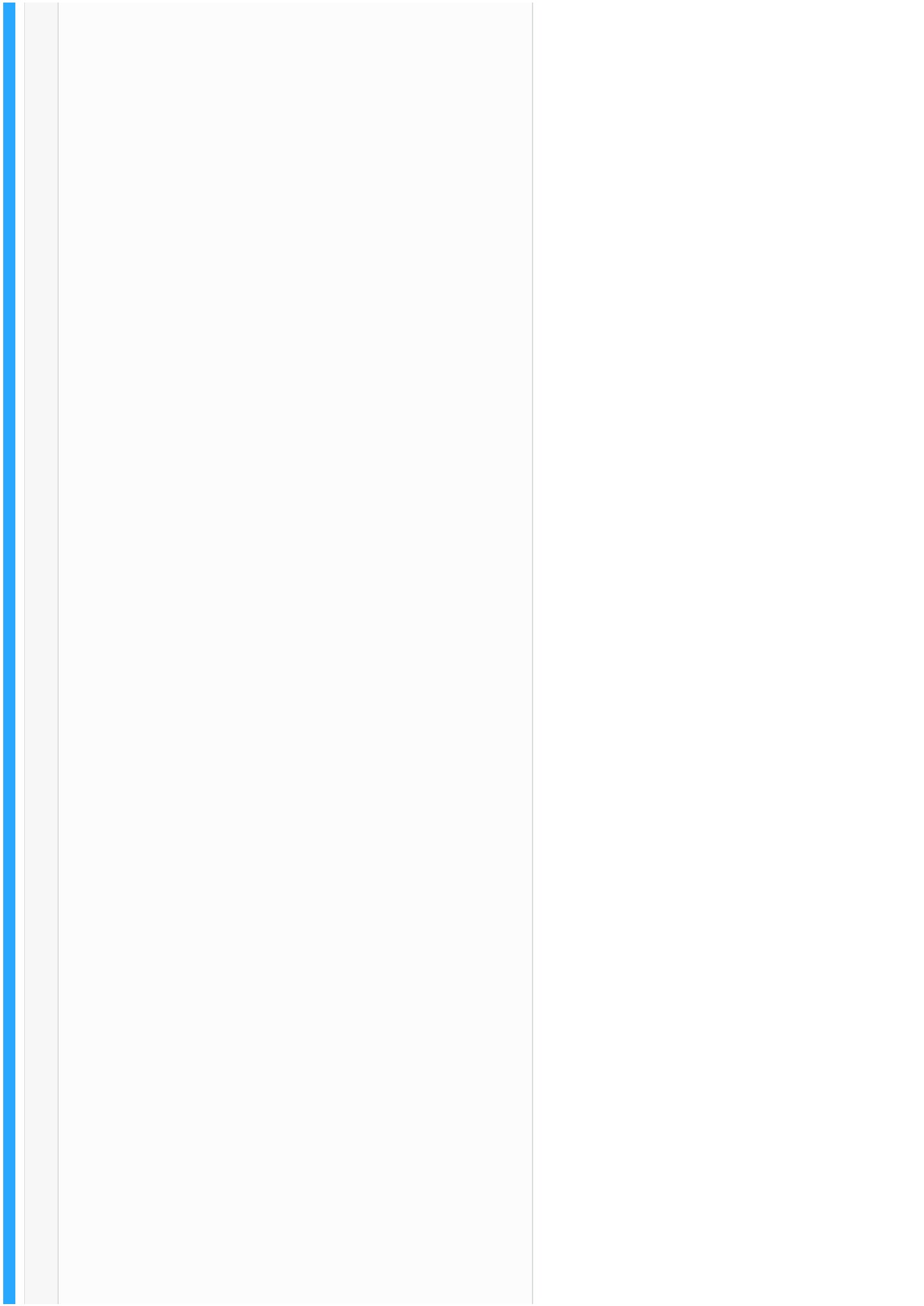


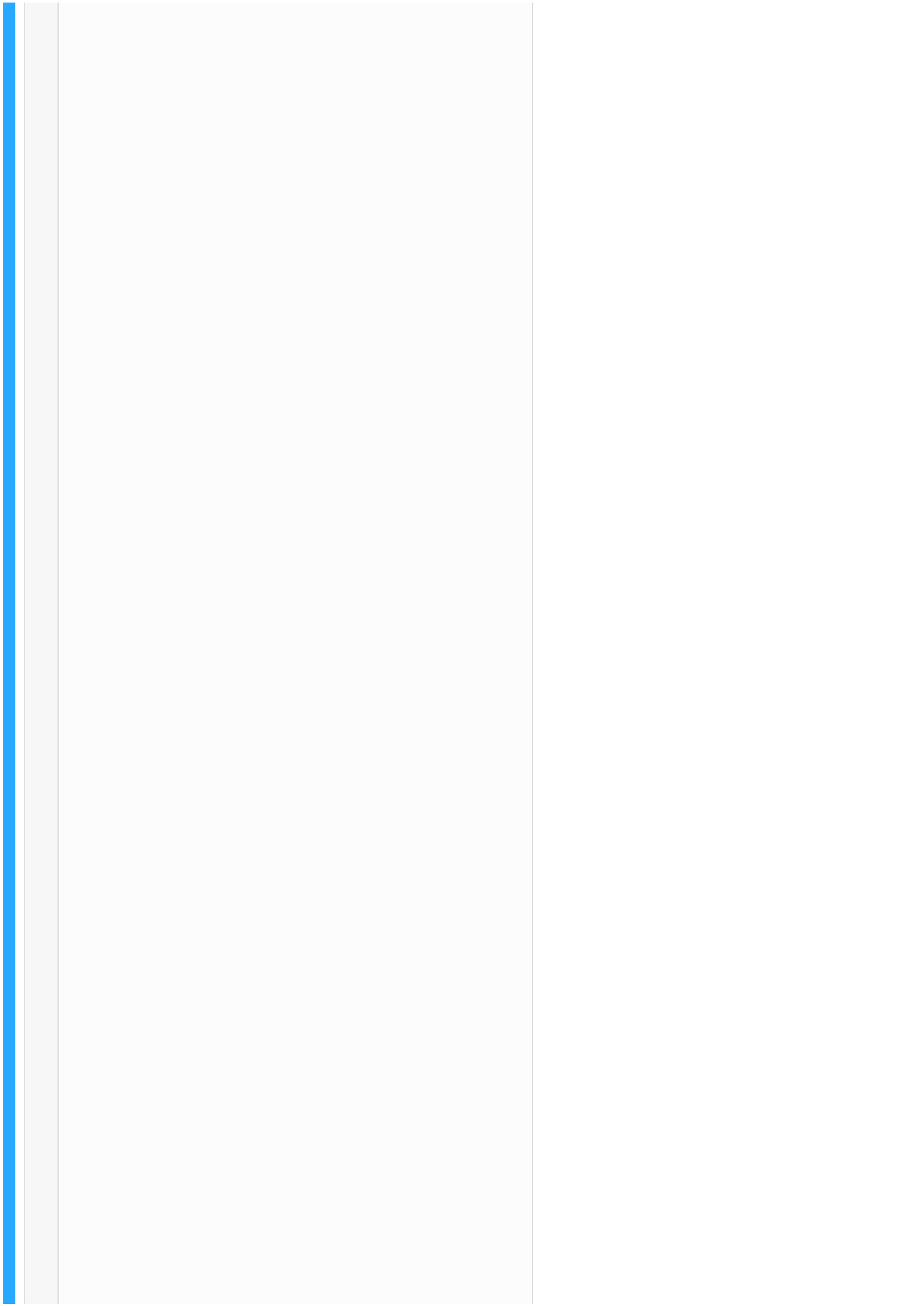


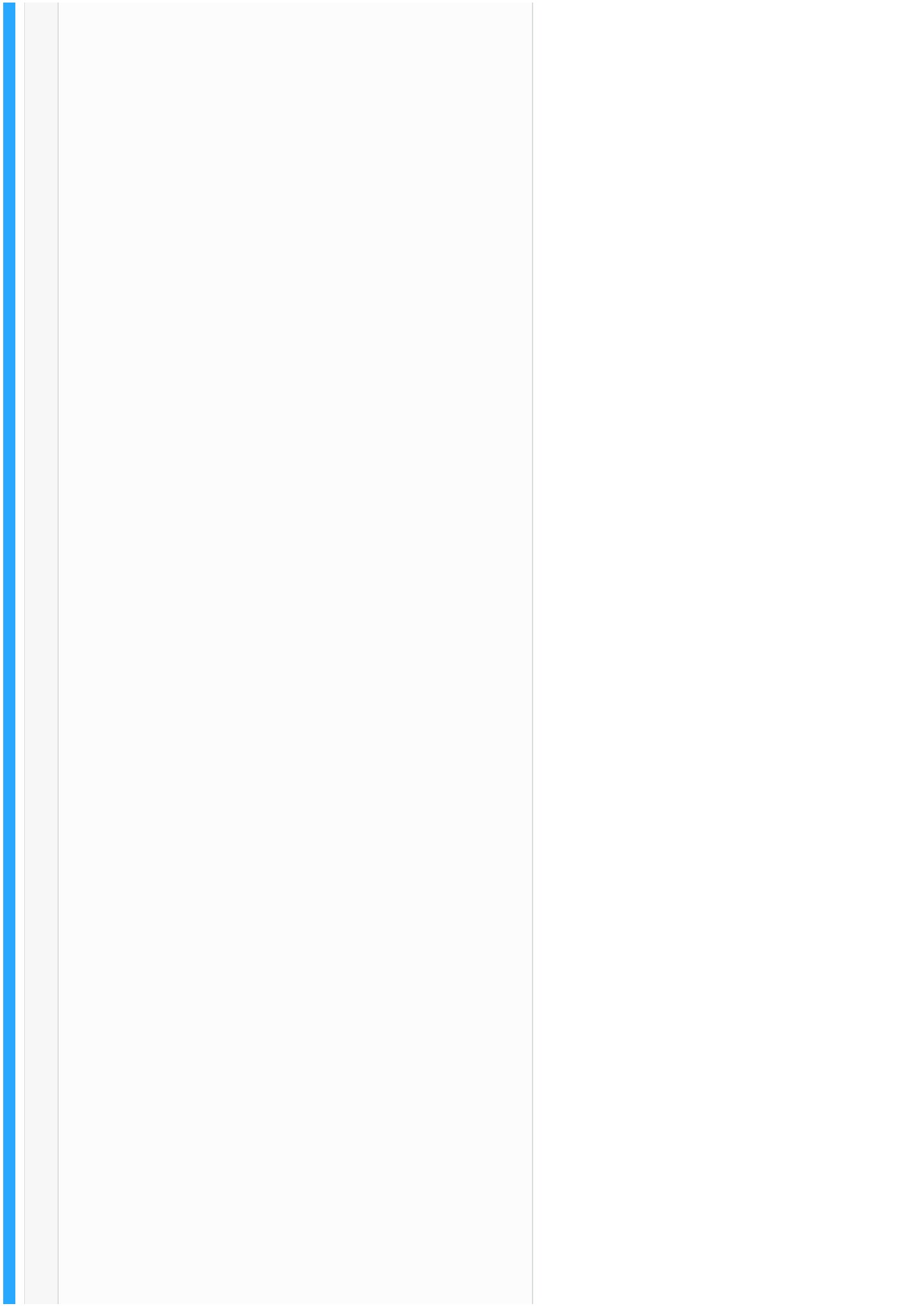


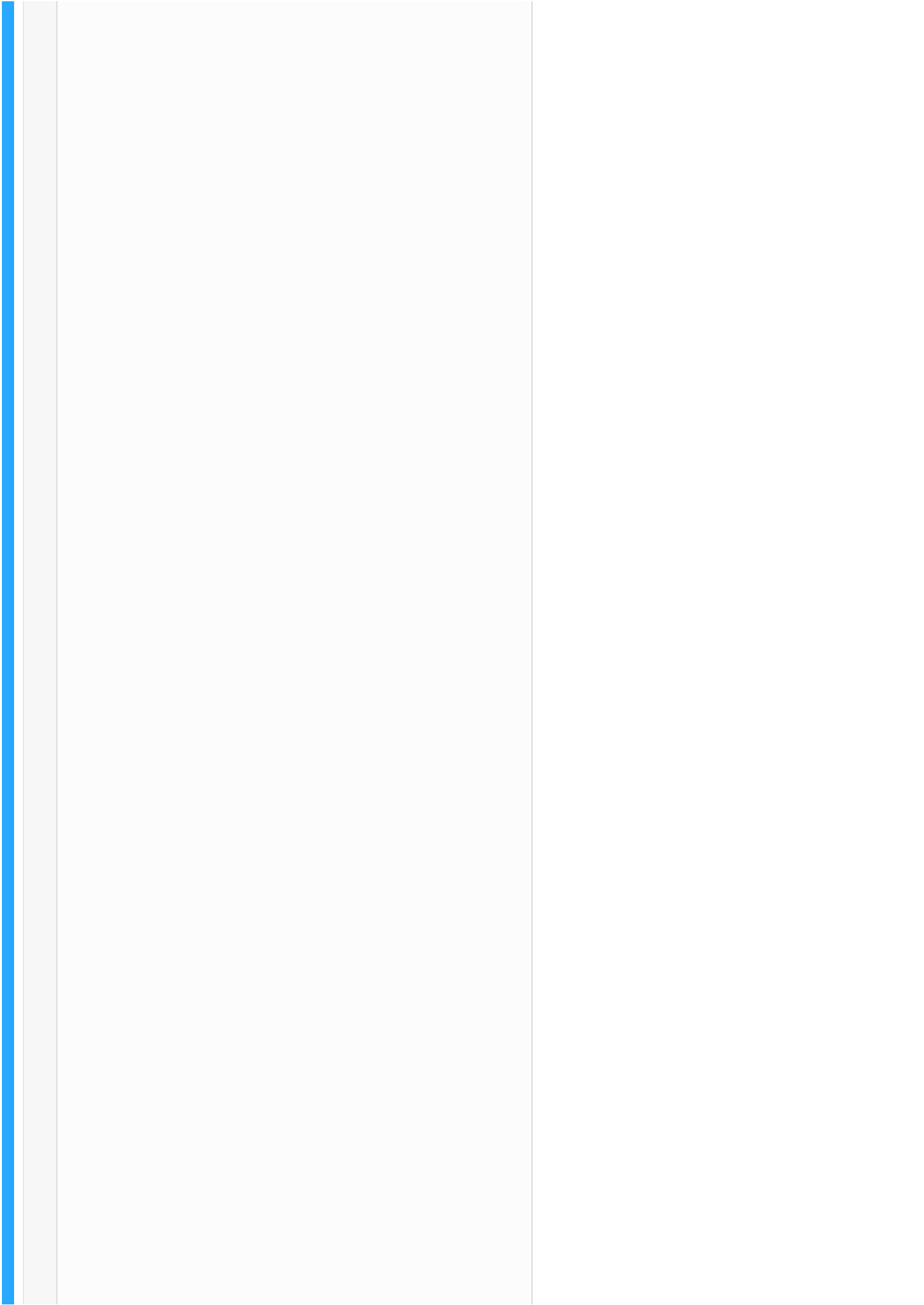


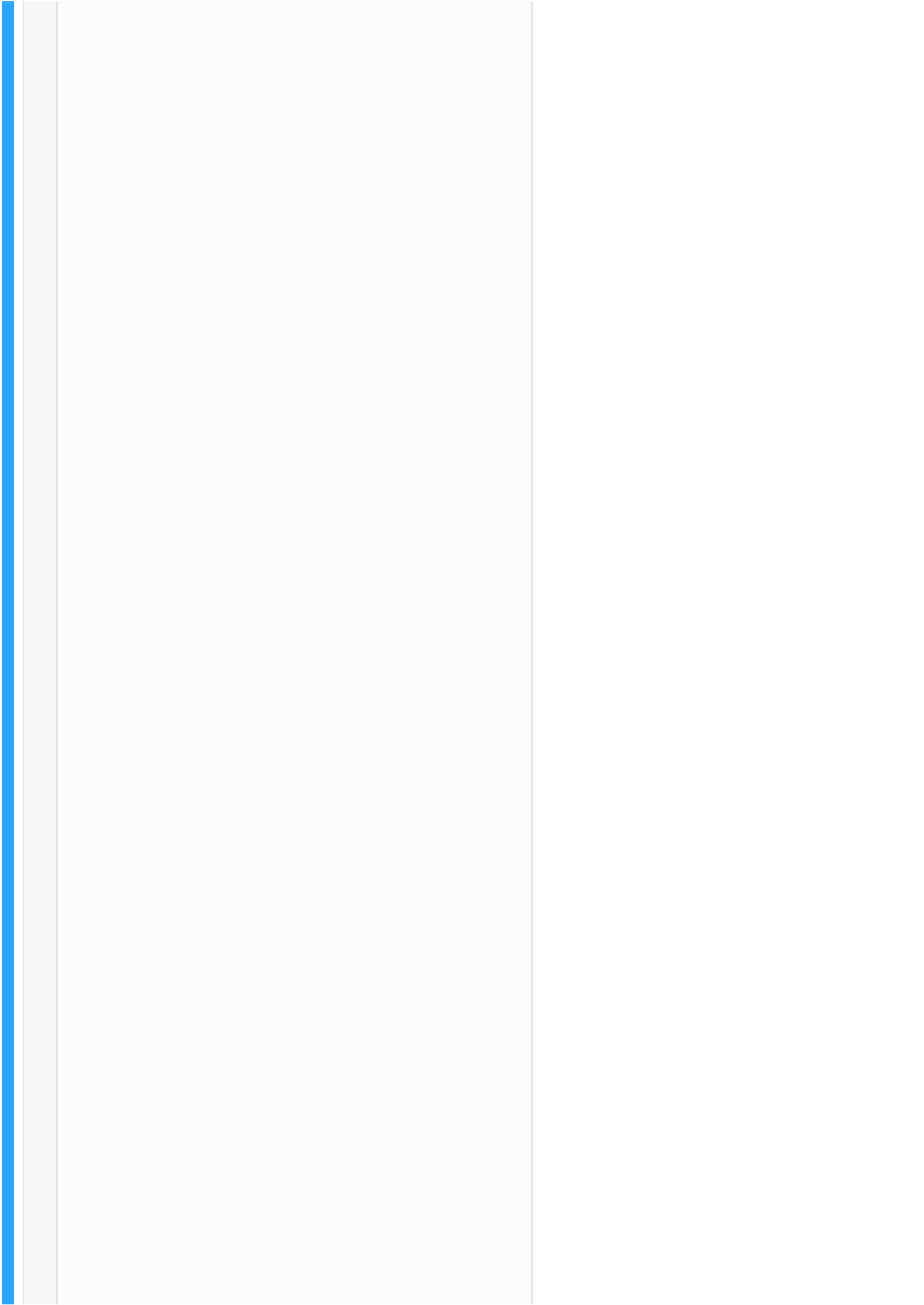




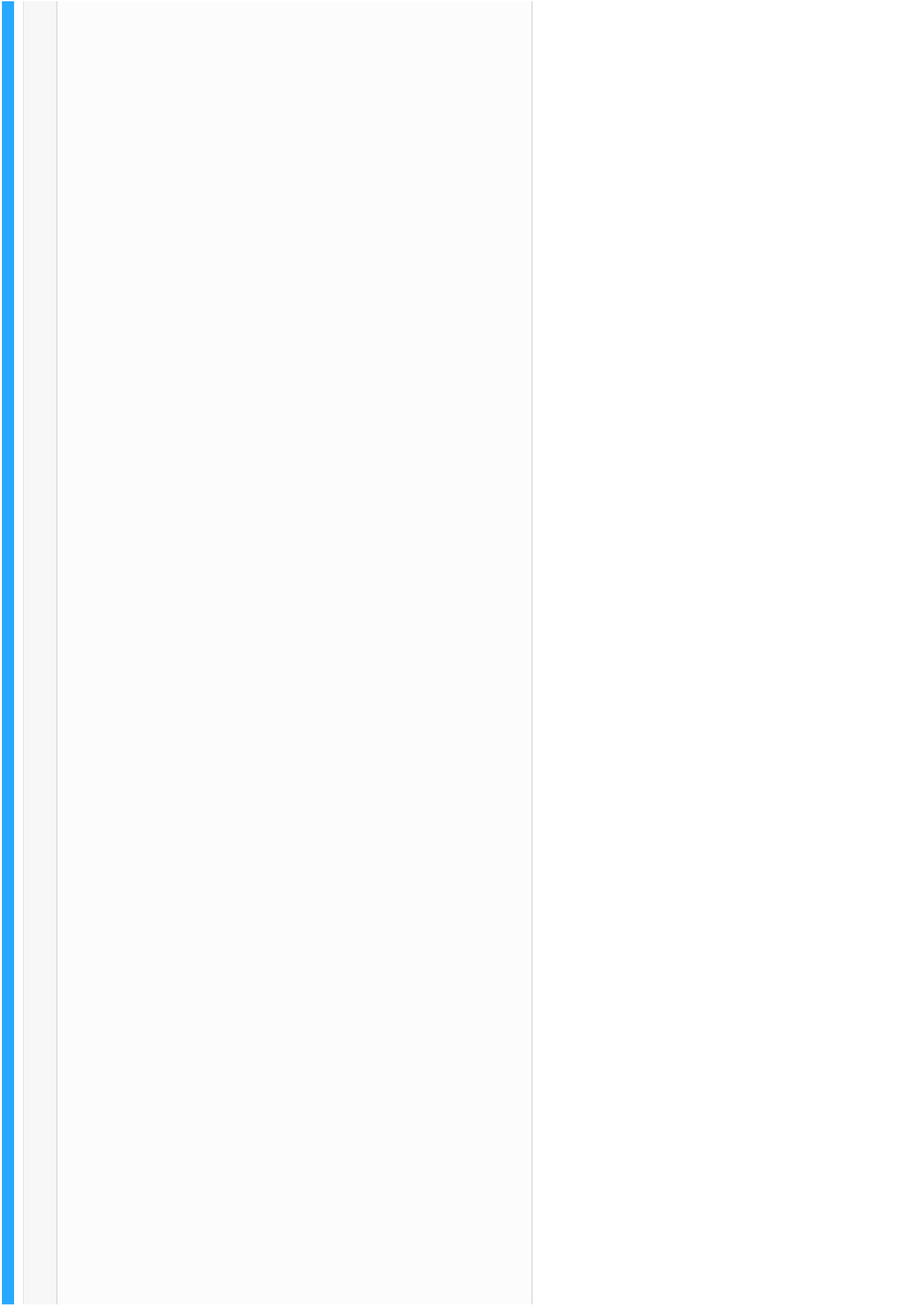


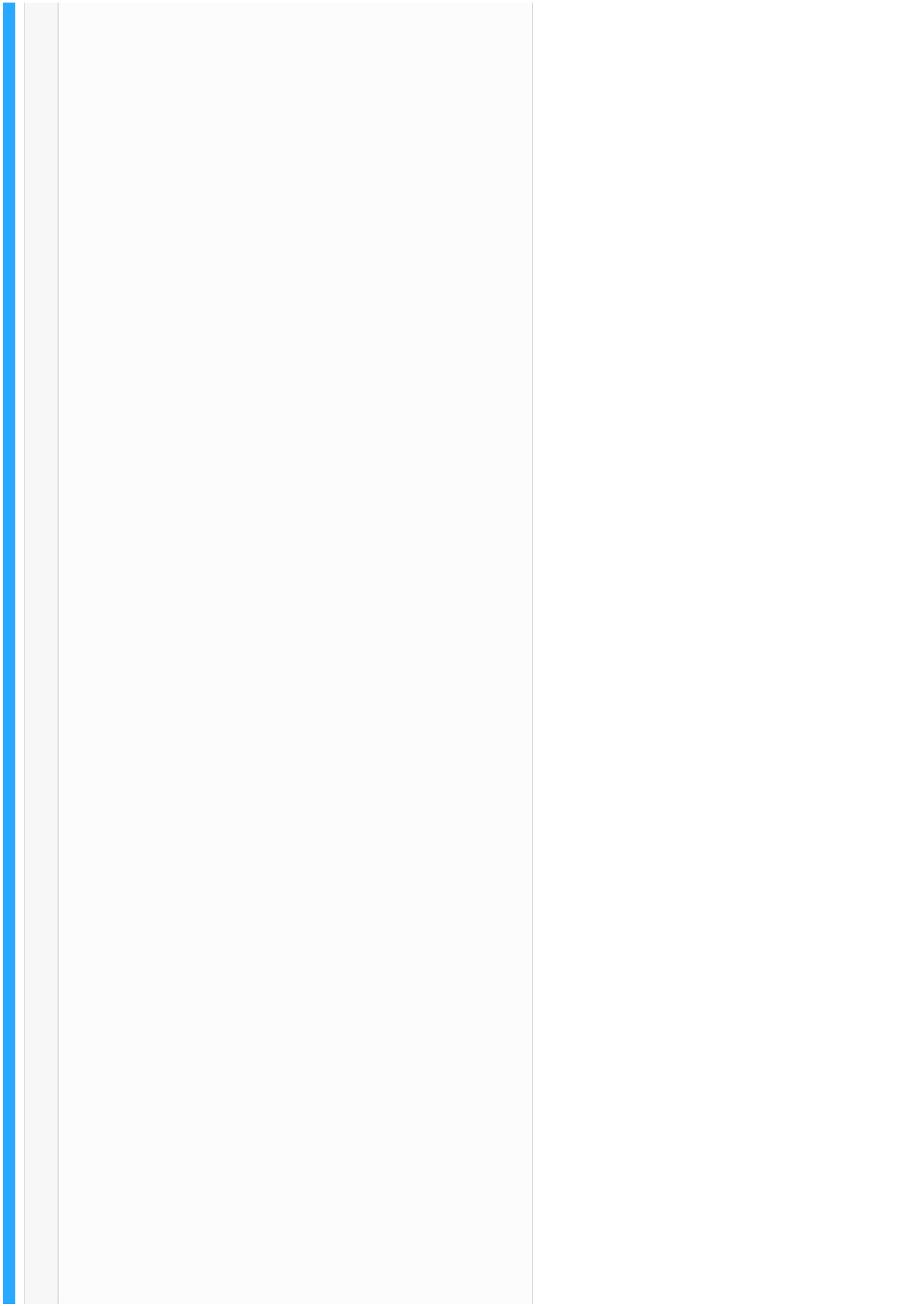


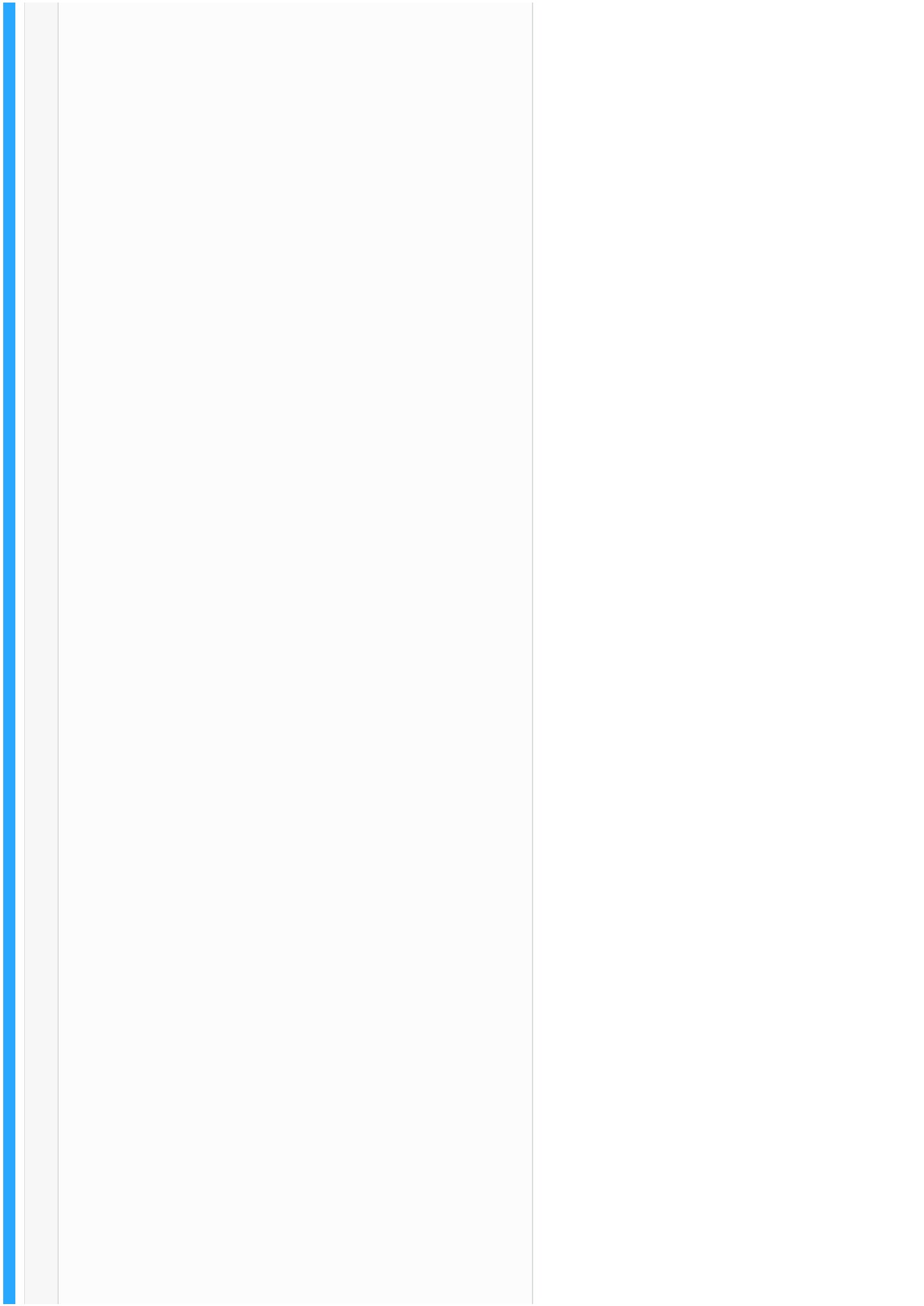














DZone

Web Dev Zone

[Log In](#) / [Sign Up](#)



[REFCARDZ](#) [GUIDES](#) [ZONES](#) | [Agile](#) [AI](#) [Big Data](#) [Cloud](#) [Database](#) [DevOps](#) [Integration](#) [IoT](#)











The code above is enough to render a simple query builder with a measure select. Here's how it looks in the CodeSandbox:

Similar to `measures`, `availableMeasures`, and `updateMeasures`, there are properties to render and manage dimensions, segments, time, filters, and chart types. [You can find the full list of properties in the documentation.](#)

Also, it is worth checking the source code of a more complicated query builder from Cube.js Playground. [You can find it on GitHub here.](#)

The New Standard for a Hybrid CMS: GraphQL Support, Scripting as a Service, SPA Support. [Watch on-demand now.](#)

Like This Article? Read More From DZone



Declarative D3
Charts With React
16.3 [Text and
Video]



PyDev of the Week:
Jason Myers



4 Useful JavaScript
Libraries for Data



Free DZone Refcard
Low-Code

Topics: WEB DEV, REACT.JS, CUBE.JS, DATA VISUALIZATION



Like (1)



Comment (0)



Save



Tweet



662 Views

Published at DZone with permission of Artyom Keydunov . [See the original article here.](#)

Opinions expressed by DZone contributors are their own.