



Progressive Rendering Format

The Progressive Rendering Format is an open specification that enables edge-accelerated streaming and partial rendering for any web framework. It allows you to deliver static content instantly from the edge while progressively loading dynamic content.

For an overview and background on this format, see our [blog post introducing the Progressive Rendering Format](#).

How It Works

The Progressive Rendering Format uses a structured JSON response format to define:

1. Static content that can be served immediately from edge nodes
2. Dynamic content that should be fetched and streamed progressively
3. How these components should be combined in the final response

When a request is processed through Tilda's infrastructure, our edge network:

1. Identifies Progressive Rendering responses via the `progressive-rendering-format: 1` header
2. Parses the format and begins streaming static content immediately
3. Fetches dynamic content from origin servers in parallel
4. Streams the combined response progressively to the client

Format Specification

The Progressive Rendering Format is defined as a JSON structure with the following schema:

Progressive Rendering Format Schema

JSON



```
1 {  
2   "v1": {
```

```

3     "status": 200, // HTTP status code
4     "headers": { // Headers for the final response
5         "content-type": ["text/html; charset=utf-8"]
6     },
7     "body": [ // Array of body segments
8         {
9             "text": "<!DOCTYPE html>..." // Static content (delivered immediatel
y)
10        },
11        {
12            "remoteBody": { // Dynamic content (fetched and streame
d)
13                "relativeUrl": "/api/data",
14                "method": "GET",
15                "forwardRequestHeaders": true,
16                "additionalHeaders": {}
17            }
18        },
19        {
20            "text": "</div></body></html>" // More static content
21        }
22    ]
23 }
24 }

```

Properties

- **v1**: The version of the Progressive Rendering Format
- **status**: The HTTP status code for the final response
- **headers**: HTTP headers to be applied to the final response
- **body**: An array of content segments that make up the response

Body Segments

Body segments can be one of two types:

- **Static text segments** (using the `text` property):
 - Delivered immediately from the edge
 - Cacheable at the edge for optimal performance
 - Typically contain the HTML shell, CSS, and initial JavaScript
- **Remote body segments** (using the `remoteBody` property):

- Content fetched from your origin server or APIs
- Streamed to the client as it becomes available
- Can include personalized or dynamic content

Remote Body Options

The `remoteBody` object supports the following properties:

remoteBody PropertiesJSON

```
1 {
2   "remoteBody": {
3     "relativeUrl": "/api/data",           // URL to fetch content from
4     "method": "GET",                     // HTTP method (GET, POST, etc.)
5     "forwardRequestHeaders": true,       // Forward original request headers
6     "additionalHeaders": {              // Additional headers to include
7       "custom-header": "value"
8     },
9     "body": "request body content"      // Optional body for POST/PUT requests
10  }
11 }
```

Request Variables

The format supports special variables that dynamically insert values from the original request:

Request VariablesJSON

```
1 {
2   "remoteBody": {
3     "relativeUrl": "$$requestPath$",      // Original request path
4     "method": "$$requestMethod$",        // Original request method
5     "additionalHeaders": {
6       "x-original-url": "$$requestUrl$",  // Full original URL
7       "x-original-host": "$$requestHost$" // Original host
8     }
9   }
10 }
```

Standard Request Variables

Variable	Description
<code>\$\$requestPath\$\$</code>	The path portion of the original request URL
<code>\$\$requestMethod\$\$</code>	The HTTP method of the original request
<code>\$\$requestUrl\$\$</code>	The full URL of the original request
<code>\$\$requestQuery\$\$</code>	The query string of the original request (without the ?)
<code>\$\$requestHost\$\$</code>	The host of the original request
<code>\$\$requestRelativeUrl\$\$</code>	The path and query of the original request

Next.js PPR Specific Variables

Variable	Description
<code>\$\$requestPathPrefixedWith NextPostponedResume\$\$</code>	Original path prefixed with <code> /_next/postponed/resume</code>
<code>\$\$requestRelativeUrlPrefixedWith NextPostponedResume\$\$</code>	Original path and query prefixed with <code> /_next/postponed/resume</code>

Implementing the Format

To use the Progressive Rendering Format in your application:

1. Using Next.js PPR

The easiest way to get started is with Next.js PPR, which is already integrated with the Progressive Rendering Format:

Enable PPR with Next.js	Bash	
<pre>1 tilda build nextjs --ppr=15-canary-v1</pre>		

See the [Next.js PPR documentation](#) for more details.

2. Build-time Static Implementation

For maximum performance, you can pre-generate Progressive Rendering Format JSON files at build time and configure your routing to serve them with the appropriate headers:

1. Generate Progressive Rendering Format JSON files during your build process
2. Place these files in a static directory (e.g., `.tilda/stage` or your own build directory)
3. Configure routing to serve these files with the `progressive-rendering-format: 1` header

This approach is used by the Tilda CLI's `tilda build nextjs --ppr` command to implement Next.js PPR. It:

1. Identifies routes marked for Partial Prerendering in the Next.js build output
2. Extracts the static shell HTML content and dynamic content information
3. Generates Progressive Rendering Format JSON files for each route
4. Places them in `.tilda/stage` directory
5. Configures routing to serve these files with the `progressive-rendering-format: 1` header

This same approach can be used for any framework by creating a custom build process that:

Build-time Implementation Example

JavaScript



```
1 // 1. Generate Progressive Rendering Format files during build
2 const progressiveRenderingFile = {
3   v1: {
4     status: 200,
5     headers: {
6       'content-type': ['text/html; charset=utf-8']
7     },
8     body: [
9       { text: '<html><head>...</head><body><div id="app">' },
10      {
11        remoteBody: {
12          relativeUrl: '/api/dynamic-data',
13          method: 'GET',
14          forwardRequestHeaders: true
15        }
16      },
17      { text: '</div></body></html>' }
18    ]
19  }
```

```

20 };
21
22 // 2. Write to files in your build directory
23 fs.writeFileSync(
24   '.tilda/stage/route-name.progressiverendering.json',
25   JSON.stringify(progressiveRenderingFile)
26 );
27
28 // 3. Configure routing (e.g., in Tilda routing config)
29 const routingConfig = {
30   routes: [{
31     criteria: { path: { exact: '/route-path' }, method: ['GET', 'HEAD'] },
32     action: {
33       origin: 'static',
34       headers: [['progressive-rendering-format', '1'], ['content-type', 'application/json']],
35       staticFileRelativePath: 'route-name.progressiverendering.json'
36     }
37   }]
38 };
39
40 // 4. Use the Tilda CLI to deploy with your routing config and preserved stage directory
41 // tilda build --routingConfigJson '...' --preserveStage

```

Custom implementations can generate these files in the `.tilda/stage` directory or their own build directory, and then use the `tilda build` command with appropriate routing configuration and the `--preserveStage` flag if needed.

3. Dynamic Server Implementation

For applications that need to generate the Progressive Rendering Format dynamically at request time, your server needs to:

1. Set the header `progressive-rendering-format: 1`
2. Add a `Tilda-Cache-Control` header with a large `max-age` value (e.g., `max-age=31536000`) to ensure the JSON format response is cached on the CDN
3. Return a JSON response following the Progressive Rendering Format

Important: The `Tilda-Cache-Control` header mentioned above applies to the HTTP response containing the JSON Progressive Rendering Format, not the inner response headers defined within the format itself. Proper caching of the format response is

crucial for performance as it allows the static shell to be served directly from edge nodes.

Example server-side implementation (pseudo-code):

Dynamic Server Implementation Example

JavaScript



```
1  async function handleRequest(request) {
2    // Determine which parts should be static vs. dynamic
3    const staticShell = generateStaticShell();
4    const dynamicContentUrl = '/api/dynamic-content';
5
6    // Return Progressive Rendering Format response
7    return new Response(JSON.stringify({
8      v1: {
9        status: 200,
10       headers: {
11         'content-type': ['text/html; charset=utf-8']
12       },
13       body: [
14         { text: staticShell.start },
15         {
16           remoteBody: {
17             relativeUrl: dynamicContentUrl,
18             method: 'GET',
19             forwardRequestHeaders: true
20           }
21         },
22         { text: staticShell.end }
23       ]
24     }
25   })), {
26     headers: {
27       'content-type': 'application/json',
28       'progressive-rendering-format': '1',
29       // This header applies to the Progressive Rendering Format response itself
30       // This ensures the format definition is cached at the edge
31       'tilda-cache-control': 'public, max-age=0, max-age=31536000'
32     }
33   });
34 }
```

Benefits

Implementing the Progressive Rendering Format provides several key advantages:

- **Improved Performance:** Static content is served instantly from edge nodes closest to users
- **Better Core Web Vitals:** Faster Time to First Byte (TTFB) and Largest Contentful Paint (LCP)
- **Reduced Origin Load:** Edge networks serve static content, reducing load on your application servers
- **Enhanced Scalability:** Better handling of traffic spikes and high-load scenarios
- **Personalization without Penalties:** Deliver personalized content without sacrificing initial load times

Current Limitations

Be aware of the following current limitations:

- For streaming to work properly, clients must support chunked transfer encoding
- Some older browsers may not fully benefit from the progressive nature of the content delivery

Further Resources

- [Progressive Rendering Format announcement](#)
- [Edge-Accelerated Next.js PPR on Tilda](#)
- [Next.js PPR documentation](#)

Resources	Online	Company
Pricing	X (Twitter)	Blog
Documentation	Github	Contact
	LinkedIn	Privacy Policy
		Terms of Service