# ExecutionContext

August 18, 2026

## ExecutionContext

**Kind:** Interface

**Source:** `src/context.ts`

Interface for the execution context in a web worker or similar environment.

`ExecutionContext` carries request-scoped state and worker lifecycle controls. It exposes `props` and `exports` for integration data, while `waitUntil()` and `passThroughOnException()` control asynchronous work and error handling.

### Properties

| Property | Type |
| --- | --- |
| `props` | `any` |
| `exports` | `any` |

### Diagram

```
graph LR
  Handler[Request Handler] --> Context[ExecutionContext]
  Context --> Props[props]
  Context --> Exports[exports]
  Context --> Wait[waitUntil()]
  Context --> PassThrough[passThroughOnException()]
  Wait --> AsyncWork[Background async work]
  PassThrough --> Upstream[Upstream request handling]
```

### Usage

```
function handleRequest(request: Request, ctx: ExecutionContext) {
  ctx.passThroughOnException();
```

```
  ctx.waitUntil(
    Promise.resolve().then(() => {
      console.log("Request completed:", request.url);
    }),
  );

  const config = ctx.props;
  const handlers = ctx.exports;

  return new Response(JSON.stringify({ config, handlers }));
}
```

## AI Coding Instructions

- Treat `ExecutionContext` as request-scoped state; do not retain it outside the request lifecycle.
- Use `waitUntil()` for asynchronous work that may continue after the response is returned.
- Call `passThroughOnException()` only when failed worker handling should fall back to upstream request handling.
- Access `props` and `exports` defensively because their shapes are typed as `any`.