# ddp

Source code of released version | Source code of development version

---

DDP (Distributed Data Protocol) is the stateful websocket protocol that Meteor uses to communicate between the client and the server. For more information about DDP, see the DDP project page or the DDP specification.

This package is used by nearly every Meteor application and provides a full implementation of DDP in JavaScript. API documentation is on the main Meteor documentation page, under "Publish and subscribe", "Methods", and "Server connections". Note in particular that clients can use `DDP.connect` to open a DDP connection to any DDP service on the Internet.

### Supported platforms

Currently the `ddp` package provides both the DDP client and the server in the same package. We plan to separate it out into separate `ddp-client` and `ddp-server` packages in the future. The client in `ddp` works with all supported Isobuild architectures, including browsers ( `web.browser.*` ), mobile devices ( `web.cordova.*` ), and node.js ( `os.*` ), using an appropriate transport for each. The server in `ddp` only works under node.js at present, due to the lack of a good way to run a websocket server in the browser. Maybe one day this will change with WebRTC.

### Supported transports

`ddp` supports the standard DDP transport, stringified EJSON over websockets. It also supports a second transport, EJSON over HTTP long polling. It will automatically fall back to this transport if websockets are not available. This is not as efficient as websockets, but it works in older browsers. It uses the sockjs library for long polling.

The sockjs-based long polling transport is not part of the official DDP standard. This is because, although sockjs works well, there is not a formal written standard for its behavior. It uses many different websocket emulation strategies for the widest possible browser compatibility, and standardizing all of these strategies would take a great deal of effort. Plus, websockets are widely available, so the need for this alternative transport will go away in the not-too-distant future. For now, if you are writing your own DDP implementation and want to interoperate with `ddp` on old browsers that don't have websockets, just use sockjs directly.

When using the sockjs transport, `ddp` has some special functionality to get around a limitation of long polling. Web browsers put a limit on the total number of HTTP connections that can be open to a particular domain at any one time, across all browser tabs. This is a potential problem for any app that uses long polling and that might be opened in multiple tabs. When too many tabs have been opened, the app can no longer connect to the server, since each of the tabs is holding open a long polling HTTP connection and the limit is hit. If this is an issue, `ddp` can be configured to use a randomly generated subdomain for each long polling connection. This requires an appropriate wildcard DNS record on the server and appropriate proxy configuration. Meteor wires all of this up automatically when deploying an app with `meteor deploy` .

### Notable features

**Database driver integration.** `ddp` works well with the Mongo database drivers. For example, if you are using the `mongo` package, then when you create a Mongo collection with `MyCollection = new Mongo.Collection("mycollection")` on the client, the Mongo driver will automatically register with `ddp` to receive incoming data for `mycollection` and use it to keep `MyCollection` up to date. In other words, it automatically wires up replication for all of your remote collections.

**Automatic latency compensation.** `ddp` includes a full implementation of fine-grained latency compensation, so users see their screen update instantly when they make changes, without having to wait for a server round trip. It will cooperate with any full-stack DB drivers that are in use to snapshot and restore records as necessary.

**Transparent reconnection.** If the DDP client loses its connection to the server, it will automatically reconnect, transparently to the application. Any subscriptions will be re-established and resynchronized without disturbing the application, and any outstanding method calls will be retried. However, this retrying could lead to duplicate method calls if the connection is lost after the server has received the method call, but before the client reads the result. This can be avoided just as it is with REST, by including a unique code as a parameter to the method. A future version of DDP will solve this on the protocol level (see Future Work).

**Authentication.** `ddp` 's authentication hooks work great with Meteor Accounts, a set of packages which provides a full suite of authentication functionality, from password-based accounts with email verification and password recovery, to OAuth login using services like Facebook and Twitter.

**Input sanitization.** On the server, you can use the `check` function, provided by the match package, to easily validate the types of arguments passed by the client. Using a simple pattern language, you can also `check` whether objects have the expected keps and array elements have the right type. In production code, add the audit-argument-checks package, and `ddp` will make sure that every value passed from the client is validated with `check` , and throw an exception if not. But be careful. The check only happens after the code has run, so while it will catch the vast majority of sanitization failures, it's not a perfect guarantee of safety.

**Tracker-aware.** `ddp` obeys the simple Tracker convention for transparent reactivity. Values such as the current connection status (are we online?), subscription readiness (is the `newsFeed` done loading or should we show a progress indicator?), and the currently logged-in user (what username should we show in the status area at the top of page?) all work as reactive variables.

**Default connection.** Normally you open a DDP connection with `myconn = DDP.connect(url)` , and then work with the connection with calls like `myconn.subscribe("newsFeed")` or `myconn.call("transferBalance")` . But if you build and deploy your app with the other Meteor tools, then a DDP server instance is automatically set up on the server, and each client is configured to automatically open a connection to that server on startup. You can work then with this "default connection" and "default server" with easy aliases like `Meteor.subscribe` and `Meteor.call` .

**Connection lifecycle hooks.** Servers can run code when connections are established or closed. This can be used to update the database to show which users are online, making it easily to create presence features like a live-updating "Friends Online Now" list.

**CRUD boilerplate and quickstart packages.** The full-stack database drivers provide some helpful functionality that is worth mentioning here, even though it is actually part of those database packages, not `ddp` . They provide general-purpose `create` , `update` , and `delete` methods for each database collection, so that it is not necessary to write methods for basic CRUD operations. A system of `allow` and `deny` rules is used to control what each user can do. And two "quickstart" packages are provided, for quick prototyping and to help new developers learn Meteor. The `insecure` package turns off `allow` / `deny` rule checking for the generic `create` , `update` , and `delete` methods. The `autopublish` package automatically subscribes every connected client to the full contents of every database collection.