Learn some jazz

Jazz is a new kind of database that's **distributed** across your frontend, containers, serverless functions and its own storage cloud.

It syncs structured data, files and LLM streams instantly, and looks like local reactive JSON state.

It also provides auth, orgs & teams, real-time multiplayer, edit histories, permissions, E2E encryption and offline-support out of the box.

Quickstart

Show me

Check out our tiny To Do list example to see what Jazz can do in a nutshell.

Help me understand

Follow our quickstart guide for a more detailed guide on building a simple app with Jazz.

Just want to get started?

You can use `create-jazz-app` to create a new Jazz project from one of our starter templates or example apps:

```
npx create-jazz-app@latest --api-key you@example.com
```

Copy

Using an LLM? Add our llms.txt to your context window!

> Requires at least Node.js v20. See our Troubleshooting Guide for quick fixes.

How it works

1. **Define your data** with CoValues schemas
2. **Connect to storage infrastructure** (Jazz Cloud or self-hosted)
3. **Create and edit CoValues** locally
4. **Get automatic sync and persistence** across all devices and users

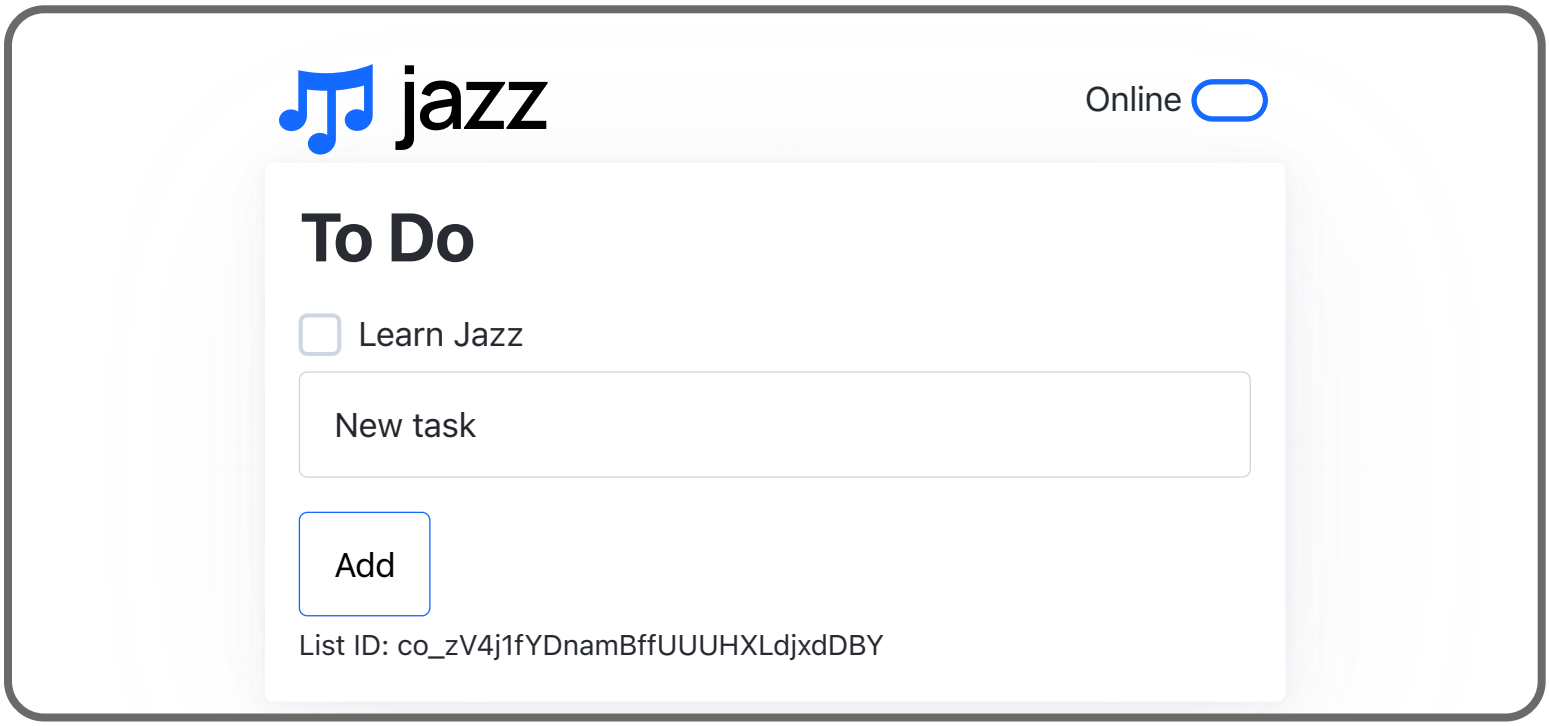
Your UI updates instantly on every change, everywhere. It's like having reactive local state that happens to be shared with the world.

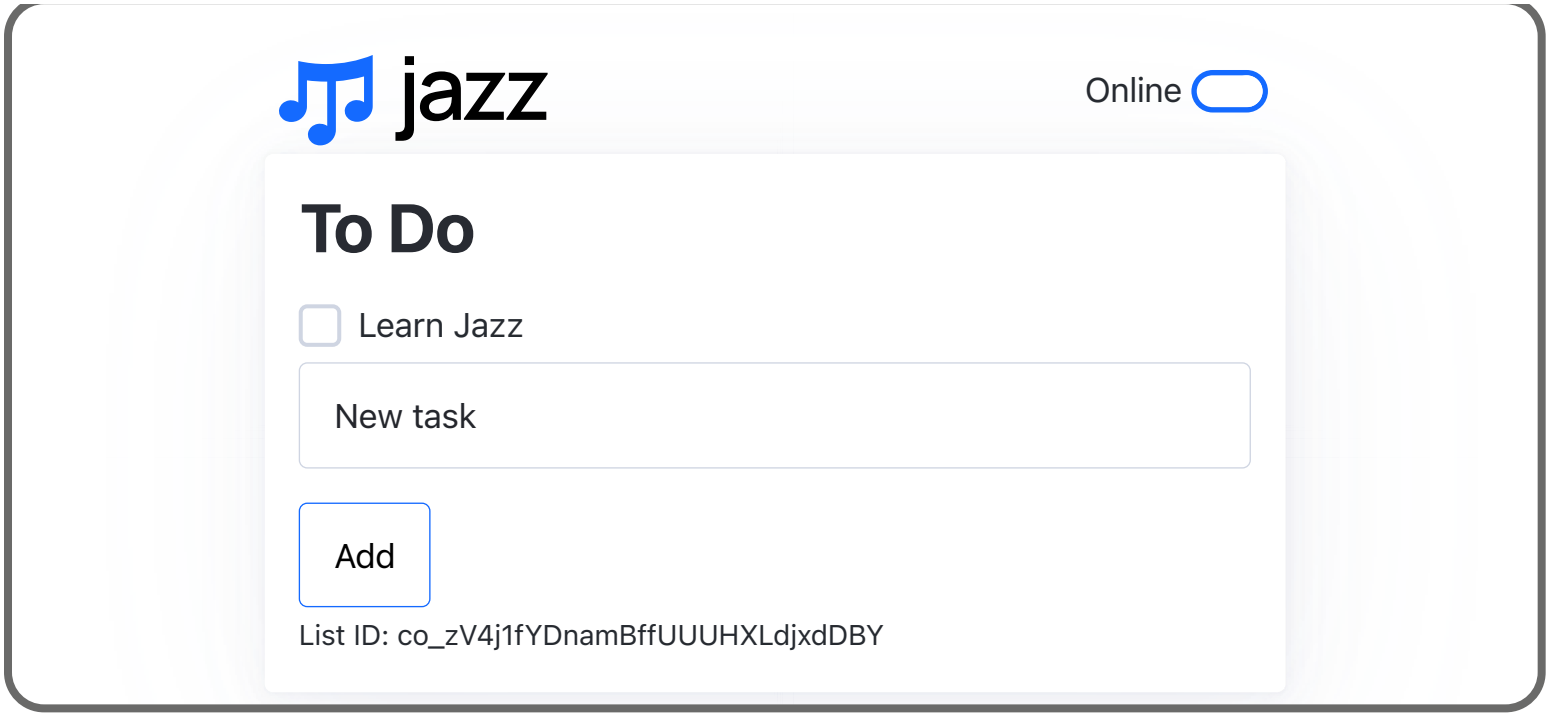
A Minimal Jazz App

Here, we'll scratch the surface of what you can do with Jazz. We'll build a quick and easy To Do list app—easy to use, easy to build, and easy to make comparisons with!

This is the end result: we're showing it here running in two iframes, updating in real-time through the Jazz Cloud.

Try adding items on the left and watch them appear instantly on the right!





> **Using Jazz Cloud**
>
> These two iframes are syncing through the Jazz Cloud. You can use the toggle in the top right to switch between 'online' and 'offline' on each client, and see how with Jazz, you can keep working even when you're offline.

Imports

Start by importing Jazz into your app.

```
import { co, z } from 'jazz-tools';
import { JazzBrowserContextManager } from 'jazz-tools/browser';
```

Copy

Schema

Then, define what your data looks like using Collaborative Values—the building blocks that make Jazz apps work.

```
const ToDo = co.map({ title: z.string(), completed: z.boolean() });
const ToDoList = co.list(ToDo);
```

Copy

Context

Next, give your app some context and tell Jazz your sync strategy—use the Jazz Cloud to get started quickly. We'll also create our to do list and get its ID here to use later.


```
await new JazzBrowserContextManager().createContext({
```

Copy

```
  sync: {
    peer: 'wss://cloud.jazz.tools?key=minimal-vanilla-example',
    when: 'always',
  },
});

const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
const listId = newList.$jazz.id;
```

Build your UI

Now, build a basic UI skeleton for your app.

```
const app = document.querySelector('#app')!;
const id = Object.assign(document.createElement('small'), {
  innerText: `List ID: ${listId}`,
});
const listContainer = document.createElement('div');
app.append(listContainer, id);
```

Display Items

Display your items and add logic to mark them as done...

```
function toDoItemElement(todo: co.loaded<typeof ToDo>) {
  const label = document.createElement('label');
  const checkbox = Object.assign(document.createElement('input'), {
    type: 'checkbox',
    checked: todo.completed,
    onclick: () => todo.$jazz.set('completed', checkbox.checked),
  });
  label.append(checkbox, todo.title);
  return label;
}
```

Add New Items

...and add new items to the list using an input and a button.

```
function newToDoFormElement(list: co.loaded<typeof ToDoList>) {
  const form = Object.assign(document.createElement('form'), {
    onsubmit: (e: Event) => {
      e.preventDefault();
      list.$jazz.push({ title: input.value, completed: false });
    }
  });
  const input = Object.assign(document.createElement('input'), {
    placeholder: 'New task',
  });
```

```
  const btn = Object.assign(document.createElement('button'), {
    innerText: 'Add',
  });
  form.append(input, btn);
  return form;
}
```

Subscribe to Changes

Now for the magic: listen to changes coming from anyone, anywhere, and update your UI in real time.

```
const unsubscribe = ToDoList.subscribe(
  listId,
  { resolve: { $each: true } },
  (toDoList) => {
    const addForm = newToDoFormElement(toDoList);
    listContainer.replaceChildren(
      ...toDoList.map((todo) => {
        return toDoItemElement(todo);
      }),
      addForm
    );
  }
);
```

Copy

Simple Routing

Lastly, we'll add a tiny bit of routing logic to be able to share the list by URL: if there's an `id` search parameter, that'll be the list we'll subscribe to later. If we don't have an `id`, we'll create a new ToDo list. We'll replace the section where we created the `ToDoList` above.

```
const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
const listId = newList.$jazz.id;

const listId = new URLSearchParams(window.location.search).get('id');

if (!listId) {
  const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
  await newList.$jazz.waitForSync();
  window.location.search = `?id=${newList.$jazz.id}`;
  throw new Error('Redirecting...');
}
```

Copy

All Together

Put it all together for a simple Jazz app in less than 100 lines of code.


```
import { co, z } from 'jazz-too
import { JazzBrowserContextManager } from 'jazz-tools/browser';
```

Copy

```
const ToDo = co.map({ title: z.string(), completed: z.boolean() });
const ToDoList = co.list(ToDo);

await new JazzBrowserContextManager().createContext({
  sync: {
    peer: 'wss://cloud.jazz.tools?key=minimal-vanilla-example',
    when: 'always',
  },
});

const listId = new URLSearchParams(window.location.search).get('id');

if (!listId) {
  const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
  await newList.$jazz.waitForSync();
  window.location.search = `?id=${newList.$jazz.id}`;
  throw new Error('Redirecting...');
}

const app = document.querySelector('#app')!;
const id = Object.assign(document.createElement('small'), {
  innerText: `List ID: ${listId}`,
});
const listContainer = document.createElement('div');
app.append(listContainer, id);

function toDoItemElement(todo: co.loaded<typeof ToDo>) {
  const label = document.createElement('label');
  const checkbox = Object.assign(document.createElement('input'), {
    type: 'checkbox',
    checked: todo.completed,
    onclick: () => todo.$jazz.set('completed', checkbox.checked),
  });
  label.append(checkbox, todo.title);
  return label;
}

function newToDoFormElement(list: co.loaded<typeof ToDoList>) {
  const form = Object.assign(document.createElement('form'), {
    onsubmit: (e: Event) => {
      e.preventDefault();
      list.$jazz.push({ title: input.value, completed: false });
    }
  });
  const input = Object.assign(document.createElement('input'), {
    placeholder: 'New task',
  });
  const btn = Object.assign(document.createElement('button'), {
    innerText: 'Add',
  });
  form.append(input, btn);
  return form;
```

```
}

const unsubscribe = ToDoList.subscribe(
  listId,
  { resolve: { $each: true } },
  (toDoList) => {
    const addForm = newToDoFormElement(toDoList);
    listContainer.replaceChildren(
      ...toDoList.map((todo) => {
        return toDoItemElement(todo);
      }),
      addForm
    );
  }
);
```

Want to see more?

Have a look at our example apps for inspiration and to see what's possible with Jazz. From real-time chat and collaborative editors to file sharing and social features—these are just the beginning of what you can build.

If you have any questions or need assistance, please don't hesitate to reach out to us on Discord. We'd love to help you get started.

Next

Quickstart

Stay up to date

Enter your email

Subscribe

About

Status

Team

Blog

Resources

Documentation

Examples

Pricing

Showcase

Learn some Jazz

Jazz is a new kind of database that's **distributed** across your frontend, containers, serverless functions and its own storage cloud.

It syncs structured data, files and LLM streams instantly, and looks like local reactive JSON state.

It also provides auth, orgs & teams, real-time multiplayer, edit histories, permissions, E2E encryption and offline-support out of the box.

#Quickstart

#Show me

Check out our tiny To Do list example to see what Jazz can do in a nutshell.

#Help me understand

Follow our quickstart guide for a more detailed guide on building a simple app with Jazz.

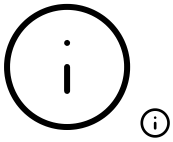
#Just want to get started?

You can use `create-jazz-app` to create a new Jazz project from one of our starter templates or example apps:

```
npx create-jazz-app@latest --api-key you@example.com
```

CopyCopied!

Using an LLM? Add our llms.txt to your context window!



Requires at least Node.js v20. See our Troubleshooting Guide for quick fixes.

#How it works

1. **Define your data** with CoValues schemas
2. **Connect to storage infrastructure** (Jazz Cloud or self-hosted)
3. **Create and edit CoValues** locally
4. **Get automatic sync and persistence** across all devices and users

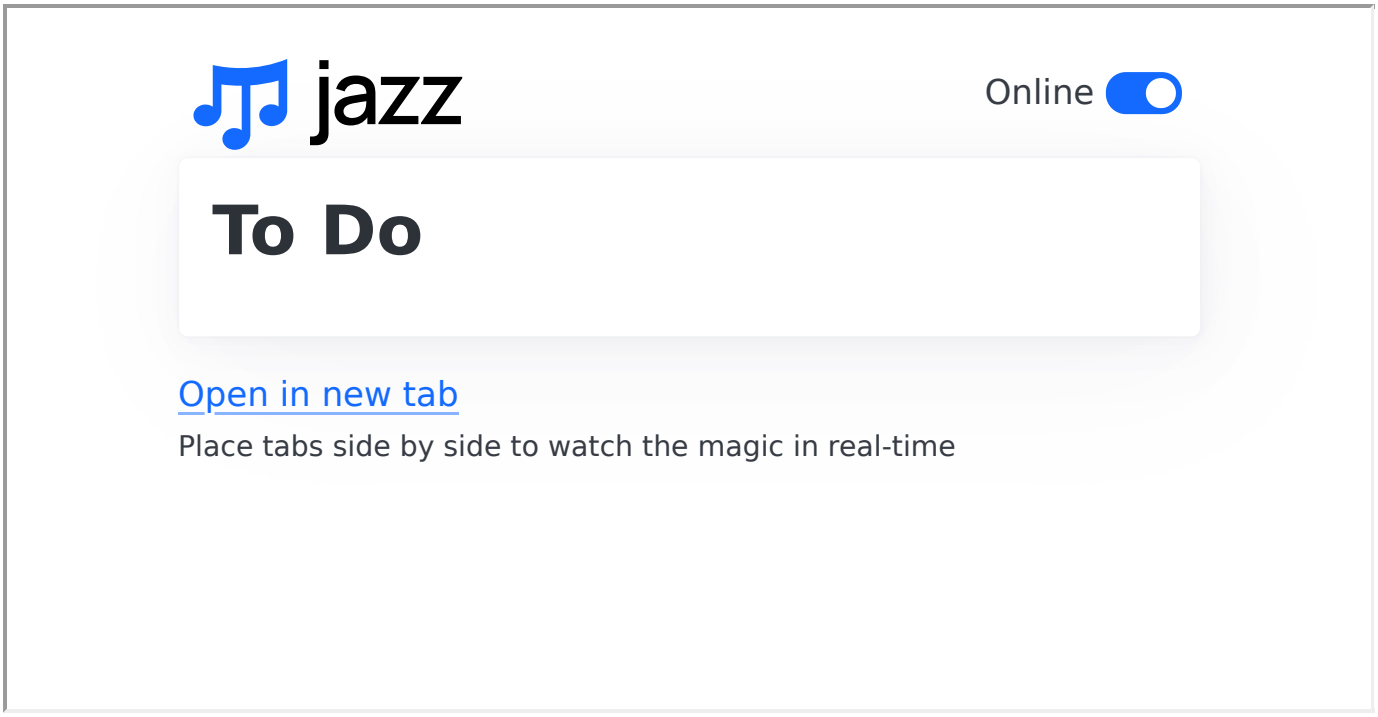
Your UI updates instantly on every change, everywhere. It's like having reactive local state that happens to be shared with the world.

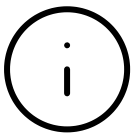
#A Minimal Jazz App

Here, we'll scratch the surface of what you can do with Jazz. We'll build a quick and easy To Do list app—easy to use, easy to build, and easy to make comparisons with!

This is the end result: we're showing it here running in two iframes, updating in real-time through the Jazz Cloud.

Try adding items on the left and watch them appear instantly on the right!





ⓘUsing Jazz Cloud

These two iframes are syncing through the Jazz Cloud. You can use the toggle in the top right to switch between 'online' and 'offline' on each client, and see how with Jazz, you can keep working even when you're offline.

#Imports

Start by importing Jazz into your app.

```
import { co, z } from 'jazz-tools';
import { JazzBrowserContextManager } from 'jazz-tools/browser';
```

CopyCopied!

#Schema

Then, define what your data looks like using Collaborative Values—the building blocks that make Jazz apps work.

```
const ToDo = co.map({ title: z.string(), completed: z.boolean() });
const ToDoList = co.list(ToDo);
```

CopyCopied!

#Context

Next, give your app some context and tell Jazz your sync strategy—use the Jazz Cloud to get started quickly. We'll also create our to do list and get its ID here to use later.

```
await new JazzBrowserContextManager().createContext({
  sync: {
    peer: 'wss://cloud.jazz.tools?key=minimal-vanilla-example',
    when: 'always',
  },
});

const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
const listId = newList.$jazz.id;
```

CopyCopied!

#Build your UI

Now, build a basic UI skeleton for your app.

```
const app = document.querySelector('#app')!;
const id = Object.assign(document.createElement('small'), {
  innerText: `List ID: ${listId}`,
});
const listContainer = document.createElement('div');
app.append(listContainer, id);
```

CopyCopied!

#Display Items

Display your items and add logic to mark them as done...

```
function toDoItemElement(todo: co.loaded<typeof ToDo>) {
  const label = document.createElement('label');
  const checkbox = Object.assign(document.createElement('input'), {
    type: 'checkbox',
    checked: todo.completed,
    onclick: () => todo.$jazz.set('completed', checkbox.checked),
  });
  label.append(checkbox, todo.title);
  return label;
}
```

CopyCopied!

#Add New Items

...and add new items to the list using an input and a button.

```
function newToDoFormElement(list: co.loaded<typeof ToDoList>) {
  const form = Object.assign(document.createElement('form'), {
    onsubmit: (e: Event) => {
      e.preventDefault();
      list.$jazz.push({ title: input.value, completed: false });
    }
  });
  const input = Object.assign(document.createElement('input'), {
    placeholder: 'New task',
  });
  const btn = Object.assign(document.createElement('button'), {
    innerText: 'Add',
  });
  form.append(input, btn);
  return form;
}
```

CopyCopied!

#Subscribe to Changes

Now for the magic: listen to changes coming from anyone, anywhere, and update your UI in real time.

```
const unsubscribe = ToDoList.subscribe(
  listId,
  { resolve: { $each: true } },
  (toDoList) => {
    const addForm = newToDoFormElement(toDoList);
    listContainer.replaceChildren(
      ...toDoList.map((todo) => {
        return toDoItemElement(todo);
      }),
      addForm
    );
  }
);
```

CopyCopied!

#Simple Routing

Lastly, we'll add a tiny bit of routing logic to be able to share the list by URL: if there's an `id` search parameter, that'll be the list we'll subscribe to later. If we don't have an `id`, we'll create a new ToDo list. We'll replace the section where we created the `ToDoList` above.

```
const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
const listId = newList.$jazz.id;

const listId = new URLSearchParams(window.location.search).get('id');

if (!listId) {
  const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
  await newList.$jazz.waitForSync();
  window.location.search = `?id=${newList.$jazz.id}`;
  throw new Error('Redirecting...');
}
```

CopyCopied!

#All Together

Put it all together for a simple Jazz app in less than 100 lines of code.

```
import { co, z } from 'jazz-tools';
import { JazzBrowserContextManager } from 'jazz-tools/browser';

const ToDo = co.map({ title: z.string(), completed: z.boolean() });
const ToDoList = co.list(ToDo);

await new JazzBrowserContextManager().createContext({
  sync: {
    peer: 'wss://cloud.jazz.tools?key=minimal-vanilla-example',
    when: 'always',
  },
});

const listId = new URLSearchParams(window.location.search).get('id');

if (!listId) {
  const newList = ToDoList.create([{ title: 'Learn Jazz', completed: false }]);
  await newList.$jazz.waitForSync();
  window.location.search = `?id=${newList.$jazz.id}`;
  throw new Error('Redirecting...');
}

const app = document.querySelector('#app')!;
const id = Object.assign(document.createElement('small'), {
  innerText: `List ID: ${listId}`,
});
const listContainer = document.createElement('div');
app.append(listContainer, id);

function toDoItemElement(todo: co.loaded<typeof ToDo>) {
  const label = document.createElement('label');
  const checkbox = Object.assign(document.createElement('input'), {
    type: 'checkbox',
    checked: todo.completed,
    onclick: () => todo.$jazz.set('completed', checkbox.checked),
  });
  label.append(checkbox, todo.title);
  return label;
}

function newToDoFormElement(list: co.loaded<typeof ToDoList>) {
  const form = Object.assign(document.createElement('form'), {
    onsubmit: (e: Event) => {
      e.preventDefault();
      list.$jazz.push({ title: input.value, completed: false });
    }
  });
  const input = Object.assign(document.createElement('input'), {
    placeholder: 'New task',
  });
  const btn = Object.assign(document.createElement('button'), {
    innerText: 'Add',
  });
  form.append(input, btn);
  return form;
}

const unsubscribe = ToDoList.subscribe(
  listId,
  { resolve: { $each: true } },
  (toDoList) => {
    const addForm = newToDoFormElement(toDoList);
    listContainer.replaceChildren(
```

```
      ...toDoList.map((todo) => {
        return toDoItemElement(todo);
      }),
      addForm
    );
  }
);
```

CopyCopied!

#Want to see more?

Have a look at our example apps for inspiration and to see what's possible with Jazz. From real-time chat and collaborative editors to file sharing and social features—these are just the beginning of what you can build.

If you have any questions or need assistance, please don't hesitate to reach out to us on Discord. We'd love to help you get started.

Was this page helpful?

Yes No

Docs issue? Join Discord

Next

Quickstart ›

On this page

Docs issue? Join Discord

Menu Docs Outline

Garden Computing Homepage

Computers are magic.
Time to make them less complex.

Stay up to date

Email address

Enter your email Subscribe

About

StatusTeam↗Blog↗

Resources

DocumentationExamplesPricingShowcase

GithubDiscordBlueSkyX

Subscriptions & Deep Loading

Jazz's Collaborative Values (such as CoMaps or CoLists) are reactive. You can subscribe to them to automatically receive updates whenever they change, either locally or remotely.

You can also use subscriptions to load CoValues *deeply* by resolving nested values. You can specify exactly how much data you want to resolve and handle loading states and errors.

You can load and subscribe to CoValues in one of two ways:

- **shallowly**—all of the primitive fields are available (such as strings, numbers, dates), but the references to other CoValues are not loaded
- **deeply**—some or all of the referenced CoValues have been loaded

> **Tip**
>
> Jazz automatically deduplicates loading. If you subscribe to the same CoValue multiple times in your app, Jazz will only fetch it once. That means you don't need to deeply load a CoValue *just in case* a child component might need its data, and you don't have to worry about tracking every possible field your app needs in a top-level query. Instead, pass the CoValue ID to the child component and subscribe there—Jazz will only load what that component actually needs.

Subscription Hooks

On your front-end, using a subscription hook is the easiest way to manage your subscriptions. The subscription and related clean-up is handled automatically, and you can use your data like any other piece of state in your app.

Subscribe to CoValues

The `useCoState` hook allows you to reactively subscribe to CoValues in your React components. It will subscribe to updates when the component mounts and unsubscribe when it unmounts, ensuring your UI stays in sync and avoiding memory leaks.

React Svelte

```
import { useCoState } from "jazz-tools/react";

function ProjectView({ projectId: { projectId: string } }) {
```

Copy

```
  // Subscribe to a project and resolve its tasks
  const project = useCoState(Project, projectId, {
    resolve: { tasks: { $each: true } }, // Tell Jazz to load each task in the list
  });

  if (!project.$isLoaded) {
    switch (project.$jazz.loadingState) {
      case "unauthorized":
        return "Project not accessible";
      case "unavailable":
        return "Project not found";
      case "loading":
        return "Loading project...";
    }
  }

  return (
    <div>
      <h1>{project.name}</h1>
      <ul>
        {project.tasks.map((task) => (
          <li key={task.$jazz.id}>{task.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

Note: If you don't need to load a CoValue's references, you can choose to load it *shallowly* by omitting the resolve query.

Subscribe to the current user's account

`useAccount` is similar to `useCoState`, but it returns the current user's account. You can use this at the top level of your app to subscribe to the current user's account profile and root.

React Svelte

Copy

```
import { useAccount } from "jazz-tools/react";
import { MyAppAccount } from "./schema";

function ProjectList() {
  const me = useAccount(MyAppAccount, {
    resolve: { profile: true },
  });

  if (!me.$isLoaded) {
    return "Loading...";
  }

  return (
```

```
<div>
  <h1>{me.profile.name}'s
</div>
```

```
  );
}
```

Loading States

When you load or subscribe to a CoValue through a hook (or directly), it can be either:

- **Loaded** → The CoValue has been successfully loaded and all its data is available
- **Not Loaded** → The CoValue is not yet available

You can use the `$isLoaded` field to check whether a CoValue is loaded. For more detailed information about why a CoValue is not loaded, you can check `$jazz.loadingState`:

- `"loading"` → The CoValue is still being fetched
- `"unauthorized"` → The current user doesn't have permission to access this CoValue
- `"unavailable"` → The CoValue couldn't be found or an error (e.g. a network timeout) occurred while loading

See the examples above for practical demonstrations of how to handle these three states in your application.

Suspense Hooks

`useSuspenseCoState` and `useSuspenseAccount` are the suspense-enabled counterparts for `useCoState` and `useAccount`. They integrate with React's Suspense API by suspending component rendering while the requested value loads.

Once the component renders successfully, all of the data requested is guaranteed to be loaded and accessible. In case the requested data **can't** be loaded, the hook will throw an error during rendering.

To handle the various loading states, instead of checking `$isLoaded` or `$jazz.loadingState`, these hooks should be combined with `<Suspense>` and error boundaries.

```
import { useSuspenseCoState } from "jazz-tools/react";

function ProjectViewSuspense({ projectId }: { projectId: string }) {
  // Subscribe to a project and resolve its tasks
  const project = useSuspenseCoState(Project, projectId, {
    resolve: { tasks: { $each: true } }, // Tell Jazz to load each task in the list
  });

  // We don't need to validate the loading state any more
  // useSuspenseCoState cannot return anything other than a loaded CoValue.
  if (!project.$isLoaded) {
    switch (project.$jazz.loadingState) {
      case "unauthorized":
```

```
        return "Project not accessible";
      case "unavailable":
        return "Project not found";
      case "loading":
        return "Loading project...";
    }
  }

  return (
    <div>
      <h1>{project.name}</h1>
      <ul>
        {project.tasks.map((task) => (
          <li key={task.$jazz.id}>{task.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

Check out an example error boundary here.

Deep Loading

When you're working with related CoValues (like tasks in a project), you often need to load nested references as well as the top-level CoValue.

This is particularly the case when working with CoMaps that refer to other CoValues or CoLists of CoValues. You can use `resolve` queries to tell Jazz what data you need to use.

Using Resolve Queries

A `resolve` query tells Jazz how deeply to load data for your app to use. We can use `true` to tell Jazz to shallowly load the tasks list here. Note that this does *not* cause the tasks themselves to load, just the CoList that holds the tasks.

Copy

```
const Task = co.map({
  title: z.string(),
  description: co.plainText(),
  get subtasks() {
    return co.list(Task);
  },
});

const Project = co.map({
  name: z.string(),
  tasks: co.list(Task),
});
```

```ts
const project = await Project.load(projectId);
if (!project.$isLoaded) throw new Error("Project not found or not accessible");

// This will be loaded
project.name; // string

// This *may not be loaded*, and *may not be accessible*
project.tasks; // MaybeLoaded<ListOfTasks>

const projectWithTasksShallow = await Project.load(projectId, {
  resolve: {
    tasks: true,
  },
});
if (!projectWithTasksShallow.$isLoaded)
  throw new Error("Project not found or not accessible");

// This list of tasks will be shallowly loaded
projectWithTasksShallow.tasks; // ListOfTasks
// We can access the properties of the shallowly loaded list
projectWithTasksShallow.tasks.length; // number
// This *may not be loaded*, and *may not be accessible*
projectWithTasksShallow.tasks[0]; // MaybeLoaded<Task>
```

We can use an `$each` expression to tell Jazz to load the items in a list.

```ts
const projectWithTasks = await Project.load(projectId, {
  resolve: {
    tasks: {
      $each: true,
    },
  },
});
if (!projectWithTasks.$isLoaded)
  throw new Error("Project not found or not accessible");

// The task will be loaded
projectWithTasks.tasks[0]; // Task
// Primitive fields are always loaded
projectWithTasks.tasks[0].title; // string
// References on the Task may not be loaded
projectWithTasks.tasks[0].subtasks; // MaybeLoaded<ListOfTasks>
// CoTexts are CoValues too
projectWithTasks.tasks[0].description; // MaybeLoaded<CoPlainText>
```


We can also build a query that *deeply resolves* to multiple levels:

```ts
const projectDeep = await Project.load(projectId, {
  resolve: {
    tasks: {
      $each: {
```

```
        subtasks: {
          $each: true,
        },
        description: true,
      },
    },
  },
});
if (!projectDeep.$isLoaded)
  throw new Error("Project not found or not accessible");

// Primitive fields are always loaded
projectDeep.tasks[0].subtasks[0].title; // string

// The description will be loaded as well
projectDeep.tasks[0].description; // CoPlainText
```

> ⚠ **Always load data explicitly**
>
> If you access a reference that wasn't included in your `resolve` query, you may find that it is already loaded, potentially because some other part of your app has already loaded it. **You should not rely on this.**
>
> Expecting data to be there which is not explicitly included in your `resolve` query can lead to subtle, hard-to-diagnose bugs. Always include every nested CoValue you need to access in your `resolve` query.

Where To Use Resolve Queries

The syntax for resolve queries is shared throughout Jazz. As well as using them in `load` and `subscribe` method calls, you can pass a resolve query to a front-end hook.

React Svelte

```
const projectId = "";
const projectWithTasksShallow = useCoState(Project, projectId, {
  resolve: {
    tasks: true,
  },
});
```

Copy

You can also specify resolve queries at the schema level, using the `.resolved()` method. These queries will be used when loading CoValues from that schema (if no resolve query is provided by the user) and in types defined with co.loaded.

```
const TaskWithDescription = Task.resolved({
  description: true,
});
const ProjectWithTasks = Project.resolved({
  tasks: {
    // Use `.resolveQuery` to get the resolve query from a schema and compose it in other
```

Copy

```
    $each: TaskWithDescription.resolveQuery,
  }
});

// .load() will use the resolve query from the schema
const project = await ProjectWithTasks.load(projectId);
if (!project.$isLoaded) throw new Error("Project not found or not accessible");
// Both the tasks and the descriptions are loaded
project.tasks[0].description; // CoPlainText
```

Loading Errors

A load operation will be successful **only** if all references requested (both optional and required) could be successfully loaded. If any reference cannot be loaded, the entire load operation will return a not-loaded CoValue to avoid potential inconsistencies.

```
// If permissions on description are restricted:
const task = await Task.load(taskId, {
  resolve: { description: true },
});
task.$isLoaded; // false
task.$jazz.loadingState; // "unauthorized"
```

Copy

This is also true if **any** element of a list is inaccessible, even if all the others can be loaded.

```
// One task in the list has restricted permissions
const projectWithUnauthorizedTasks = await Project.load(projectId, {
  resolve: { tasks: { $each: true } },
});

project.$isLoaded; // false
project.$jazz.loadingState; // "unauthorized"
```

Copy

Loading will be successful if all requested references are loaded. Non-requested references may or may not be available.

```
// One task in the list has restricted permissions
const shallowlyLoadedProjectWithUnauthorizedTasks = await Project.load(
  projectId,
  {
    resolve: true,
  },
);
if (!project.$isLoaded) throw new Error("Project not found or not accessible");

// Assuming the user has permissions on the project, this load will succeed, even if the
user cannot load one of the ta
project.$isLoaded; // true
// Tasks may not be loaded since we didn't request them
```

Copy

```
project.tasks.$isLoaded; // may be false
```

Catching loading errors

We can use `$onError` to handle cases where some data you have requested is inaccessible, similar to a `try...catch` block in your query.

For example, in case of a `project` (which the user can access) with three `task` items:

Task	User can access `task`?	User can access `task.description`?
0	✅	✅
1	✅	❌
2	❌	❌

Scenario 1: Skip Inaccessible List Items

If some of your list items may not be accessible, you can skip loading them by specifying `$onError: 'catch'`. Inaccessible items will be not-loaded CoValues, while accessible items load properly.

Copy

```
// Inaccessible tasks will not be loaded, but the project will
const projectWithInaccessibleSkipped = await Project.load(projectId, {
  resolve: { tasks: { $each: { $onError: "catch" } } },
});

if (!project.$isLoaded) {
  throw new Error("Project not found or not accessible");
}

if (!project.tasks.$isLoaded) {
  throw new Error("Task List not found or not accessible");
}

project.tasks[0].$isLoaded; // true
project.tasks[1].$isLoaded; // true
project.tasks[2].$isLoaded; // false (caught by $onError)
```

Scenario 2: Handling Inaccessible Nested References

An `$onError` applies only in the block where it's defined. If you need to handle multiple potential levels of error, you can nest `$onError` handlers.

This load will fail, because the `$onError` is defined only for the `task.description`, not for failures in loading the `task` itself.

```
// Inaccessible tasks will not be loaded, but the project will
const projectWithNestedInaccessibleSkipped = await Project.load(projectId, {
  resolve: {
    tasks: {
      $each: {
        description: true,
        $onError: "catch",
      },
    },
  },
});

if (!project.$isLoaded) {
  throw new Error("Project not found or not accessible");
}

project.tasks[0].$isLoaded; // true
project.tasks[1].$isLoaded; // true
project.tasks[2].$isLoaded; // false (caught by $onError)
```

Copy

We can fix this by adding handlers at both levels

```
const projectWithMultipleCatches = await Project.load(projectId, {
  resolve: {
    tasks: {
      $each: {
        description: { $onError: "catch" }, // catch errors loading task descriptions
        $onError: "catch", // catch errors loading tasks too
      },
    },
  },
});

project.$isLoaded; // true
project.tasks[0].$isLoaded; // true
project.tasks[0].description.$isLoaded; // true
project.tasks[1].$isLoaded; // true
project.tasks[1].description.$isLoaded; // false (caught by the inner handler)
project.tasks[2].$isLoaded; // false (caught by the outer handler)
```

Copy

Type safety with co.loaded

You can tell your application how deeply your data is loaded by using the `co.loaded` type.

The `co.loaded` type is especially useful when passing data between components, because it allows TypeScript to check at compile time whether data your application depends is properly loaded. The second argument lets you pass a `resolve` query to specify how deeply your data is loaded.

```
import { co } from "jazz-tools";
import { Project } from "./schema";

type ProjectWithTasks = co.loaded<
  typeof Project,
  {
    tasks: {
      $each: true;
    };
  }
>;

// In case the project prop isn't loaded as required, TypeScript will warn
function TaskList({ project }: { project: ProjectWithTasks }) {
  // TypeScript knows tasks are loaded, so this is type-safe
  return (
    <ul>
      {project.tasks.map((task) => (
        <li key={task.$jazz.id}>{task.title}</li>
      ))}
    </ul>
  );
}
```

Copy

You can pass a `resolve` query of any complexity to `co.loaded`.

Manual subscriptions

If you have a CoValue's ID, you can subscribe to it anywhere in your code using `CoValue.subscribe()`.

Note: Manual subscriptions are best suited for vanilla JavaScript—for example in server-side code or tests. Inside front-end components, we recommend using a subscription hook.

```
// Subscribe by ID
const unsubscribe = Task.subscribe(taskId, {}, (updatedTask) => {
  console.log("Updated task:", updatedTask);
});

// Always clean up when finished
unsubscribe();
```

Copy

You can also subscribe to an existing CoValue instance using the `$jazz.subscribe` method.

```
const myTask = Task.create({
  title: "My new task",
});

// Subscribe using $jazz.subscribe
```

Copy

```
const unsubscribe = myTask.$jazz.subscribe((updatedTask) => {
  console.log("Updated task:", updatedTask);
});

// Always clean up when finished
unsubscribe();
```

Error handling for manual subscriptions

You can also pass `onUnauthorized` and `onUnavailable` handlers as options to your subscription which will run reactively whenever the CoValue you're subscribing to is unavailable, or you do not have the appropriate permissions to read it.

```
const unsubscribe = Task.subscribe(taskId, {
  onUnauthorized: (err) => console.error(err),
  onUnavailable: (err) => console.error(err)
}, (updatedTask) => {
  console.log("Updated task:", updatedTask);
});

// Always clean up when finished
unsubscribe();
```

Copy

Selectors

Sometimes, you only need to react to changes in specific parts of a CoValue. In those cases, you can provide a `select` function to specify what data you are interested in, and an optional `equalityFn` option to control re-renders.

- `select` : extract the fields you care about
- `equalityFn` : (optional) control when data should be considered equal

```
function ProjectViewWithSelector({ projectId }: { projectId: string }) {
  // Subscribe to a project
  const project = useCoState(Project, projectId, {
    resolve: {
      tasks: true,
    },
    select: (project) => {
      if (!project.$isLoaded) return {
        loadingState: project.$jazz.loadingState
      };
      return {
        name: project.name,
        taskCount: project.tas
        loadingState: project.$jazz.loadingState
```

Copy

```
    };
  },
  // Only re-render if the name or the number of tasks change
  equalityFn: (a, b) => {
    if (a.loadingState !== 'loaded' || b.loadingState !== 'loaded') return false;

    return a?.name === b?.name && a?.taskCount === b?.taskCount;
  },
});

switch (project.loadingState) {
  case "unauthorized":
    return "Project not accessible";
  case "unavailable":
    return "Project not found";
  case "loading":
    return "Loading...";
}

return (
  <div>
    <h1>{project.name}</h1>
    <small>{project.taskCount} task(s)</small>
  </div>
);
}
```

By default, the return values of the select function will be compared using `Object.is`, but you can use the `equalityFn` to add your own logic.

You can also use `useAccount` in the same way, to subscribe to only the changes in a user's account you are interested in.

```
import { useAccount } from "jazz-tools/react";

function ProfileName() {
  // Only re-renders when the profile name changes
  const profileName = useAccount(MyAppAccount, {
    resolve: {
      profile: true,
    },
    select: (account) =>
      account.$isLoaded ? account.profile.name : "Loading...",
  });

  return <div>{profileName}</div>;
}
```

Copy

Avoiding Expensive Selectors

Selector functions optimise re-renders by only updating React state if the underlying CoValue has

changed in a way that you care about.

However, the selector function itself still runs on every CoValue update, even if your `equalityFn` returns `true` and your component does not re-render. Because of this, you should **avoid doing expensive computation inside a selector**.

For expensive operations, use a lightweight selector which only tracks the minimum necessary to identify when the dependencies change, and run the expensive operations separately, wrapped in a `useMemo` hook.

This way, React can batch state updates efficiently and only recompute the expensive memoised operation if the dependencies have changed.

```
function ProjectViewWithExpensiveOperations({ projectId }: { projectId: string }) {
  const project = useCoState(Project, projectId, {
    resolve: {
      tasks: {
        $each: true,
      }
    },
    select: (project) => {
      if (!project.$isLoaded) return {
        loadingState: project.$jazz.loadingState,
        tasks: [],
        taskIds: []
      };
      return {
        name: project.name,
        tasks: project.tasks,
        loadingState: project.$jazz.loadingState
      };
    },
    equalityFn: (a, b) => {
      if (a.loadingState !== 'loaded' || b.loadingState !== 'loaded') return false;
      if (a.name !== b.name) return false;
      if (a.tasks.length !== b.tasks.length) return false;
      const aTaskIds = new Set(a.tasks.map(t => t.$jazz.id));
      return b.tasks.every(t => aTaskIds.has(t.$jazz.id));
    },
  });

  const tasksAfterExpensiveComputations = useMemo(() => {
    const sortedTasks = project.tasks.slice(0).sort(someExpensiveSortFunction);
    const groupedTasks = sortedTasks.reduce(someExpensiveReduceFunction, {});
    return groupedTasks;
  }, [project.tasks]);


  switch (project.loadingState) {
    case "unauthorized":
      return "Project not accessible";
```

Copy

```
    case "unavailable":
      return "Project not found";
    case "loading":
      return "Loading...";
  }


  return (
    <div>
      <h1>{project.name}</h1>
      <GroupedTaskDisplay tasks={tasksAfterExpensiveComputations} />
    </div>
  );
}
```

Ensuring data is loaded

In most cases, you'll have specified the depth of data you need in a `resolve` query when you first load or subscribe to a CoValue. However, sometimes you might have a CoValue instance which is not loaded deeply enough, or you're not sure how deeply loaded it is. In this case, you need to make sure data is loaded before proceeding with an operation. The `$jazz.ensureLoaded` method lets you guarantee that a CoValue and its referenced data are loaded to a specific depth (i.e. with nested references resolved):

```
async function completeAllTasks(projectId: string) {
  // Load the project
  const project = await Project.load(projectId, { resolve: true });
  if (!project.$isLoaded) return;

  // Ensure tasks are deeply loaded
  const loadedProject = await project.$jazz.ensureLoaded({
    resolve: {
      tasks: {
        $each: true,
      },
    },
  });

  // Now we can safely access and modify tasks
  loadedProject.tasks.forEach((task, i) => {
    task.$jazz.set("title", `Task ${i}`);
  });
}
```

Copy

This can be useful if you have a shallowly loaded CoValue instance, and would like to load its references deeply.

Best practices

- Load exactly what you need. Start shallow and add your nested references with care.
- Always check `$isLoaded` before accessing CoValue data. Use `$jazz.loadingState` for more detailed information.
- Use `$onError: 'catch'` at each level of your query that can fail to handle inaccessible data gracefully.
- Use selectors and an `equalityFn` to prevent unnecessary re-renders.
- Never rely on data being present unless it is requested in your `resolve` query.

Previous
0.17.0 - New image APIs

Next
Sync and storage

Stay up to date

Enter your email

Subscribe

About

Status

Team

Blog

Resources

Documentation

Examples

Pricing

Showcase

CoMaps

CoMaps are key-value objects that work like JavaScript objects. You can access properties with dot notation and define typed fields that provide TypeScript safety. They're ideal for structured data that needs type validation.

Creating CoMaps

CoMaps are typically defined with `co.map()` and specifying primitive fields using `z` (see Defining schemas: CoValues for more details on primitive fields):

```
import { co, z } from "jazz-tools";

const Project = co.map({
  name: z.string(),
  startDate: z.date(),
  status: z.literal(["planning", "active", "completed"]),
  coordinator: co.optional(Member),
});
export type Project = co.loaded<typeof Project>;
export type ProjectInitShape = co.input<typeof Project>; // type accepted by `Project.crea
```

Copy

You can create either struct-like CoMaps with fixed fields (as above) or record-like CoMaps for key-value pairs:

```
const Inventory = co.record(z.string(), z.number());
```

Copy

To instantiate a CoMap:

```
const project = Project.create({
  name: "Spring Planting",
  startDate: new Date("2025-03-15"),
  status: "planning",
});

const inventory = Inventory.create({
  tomatoes: 48,
  basil: 12,
});
```

Copy

Ownership

When creating CoMaps, you can specify ownership to control access:

```
// Create with default owner (current user)
const privateProject = Project.create({
  name: "My Herb Garden",
  startDate: new Date("2025-04-01"),
  status: "planning",
});

// Create with shared ownership
const gardenGroup = Group.create();
gardenGroup.addMember(memberAccount, "writer");

const communityProject = Project.create(
  {
    name: "Community Vegetable Plot",
    startDate: new Date("2025-03-20"),
    status: "planning",
  },
  { owner: gardenGroup },
);
```

Copy

See Groups as permission scopes for more information on how to use groups to control access to CoMaps.

Reading from CoMaps

CoMaps can be accessed using familiar JavaScript object notation:

```
console.log(project.name); // "Spring Planting"
console.log(project.status); // "planning"
```

Copy

Handling Optional Fields

Optional fields require checks before access:

```
if (project.coordinator) {
  console.log(project.coordinator.name); // Safe access
}
```

Copy

Recursive references

You can wrap references in getters. This allows you to defer evaluation until the property is accessed. This technique is particularly useful for defining circular references, including recursive (self-referencing) schemas, or mutually r

```ts
import { co, z } from "jazz-tools";

const Project = co.map({
  name: z.string(),
  startDate: z.date(),
  status: z.literal(["planning", "active", "completed"]),
  coordinator: co.optional(Member),
  get subProject() {
    return Project.optional();
  },
});

export type Project = co.loaded<typeof Project>;
```

When the recursive references involve more complex types, it is sometimes required to specify the getter return type:

```ts
const ProjectWithTypedGetter = co.map({
  name: z.string(),
  startDate: z.date(),
  status: z.literal(["planning", "active", "completed"]),
  coordinator: co.optional(Member),
  get subProjects(): co.Optional<co.List<typeof Project>> {
    return co.optional(co.list(Project));
  },
});

export type Project = co.loaded<typeof Project>;
```

Partial

For convenience Jazz provies a dedicated API for making all the properties of a CoMap optional:

```ts
const Project = co.map({
  name: z.string(),
  startDate: z.date(),
  status: z.literal(["planning", "active", "completed"]),
});

const ProjectDraft = Project.partial();

// The fields are all optional now
const project = ProjectDraft.create({});
```

Pick

You can also pick specific fields from a CoMap:


```ts
const Project = co.map({
```

```
  name: z.string(),
  startDate: z.date(),
  status: z.literal(["planning", "active", "completed"]),
});

const ProjectStep1 = Project.pick({
  name: true,
  startDate: true,
});

// We don't provide the status field
const project = ProjectStep1.create({
  name: "My project",
  startDate: new Date("2025-04-01"),
});
```

Working with Record CoMaps

For record-type CoMaps, you can access values using bracket notation:

```
const inventory = Inventory.create({
  tomatoes: 48,
  peppers: 24,
  basil: 12,
});

console.log(inventory["tomatoes"]); // 48
```

Copy

Updating CoMaps

To update a CoMap's properties, use the `$jazz.set` method:

```
project.$jazz.set("name", "Spring Vegetable Garden"); // Update name
project.$jazz.set("startDate", new Date("2025-03-20")); // Update date
```

Copy

> The `$jazz` namespace is available on all CoValues, and provides access to methods to modify and load CoValues, as well as access common properties like `id` and `owner`.

When updating references to other CoValues, you can provide both the new CoValue or a JSON object from which the new CoValue will be created.

```
const Dog = co.map({
  name: co.plainText(),
});
const Person = co.map({
  name: co.plainText(),
```

Copy

```
  dog: Dog,
});

const person = Person.create({
  name: "John",
  dog: { name: "Rex" },
});

// Update the dog field using a CoValue
person.$jazz.set("dog", Dog.create({ name: co.plainText().create("Fido") }));
// Or use a plain JSON object
person.$jazz.set("dog", { name: "Fido" });
```

When providing a JSON object, Jazz will automatically create the CoValues for you. To learn more about how permissions work in this case, refer to Ownership on implicit CoValue creation.

Type Safety

CoMaps are fully typed in TypeScript, giving you autocomplete and error checking:

```
project.$jazz.set("name", "Spring Vegetable Planting"); // ✓ Valid string
project.$jazz.set("startDate", "2025-03-15"); // ✗ Type error: expected Date
// Argument of type 'string' is not assignable to parameter of type 'Date'
```

Copy

Soft Deletion

Implementing a soft deletion pattern by using a `deleted` flag allows you to maintain data for potential recovery and auditing.

```
const Project = co.map({
  name: z.string(),
  deleted: z.optional(z.boolean()),
});
```

Copy

When an object needs to be "deleted", instead of removing it from the system, the deleted flag is set to true. This gives us a property to omit it in the future.

Deleting Properties

You can delete properties from CoMaps:

```
inventory.$jazz.delete("basil"); // Remove a key-value pair

// For optional fields in struct-like CoMaps
project.$jazz.set("coordinator", undefined); // Remove the reference
```

Copy

Running migrations on CoMaps

Migrations are functions that run when a CoMap is loaded, allowing you to update existing data to match new schema versions. Use them when you need to modify the structure of CoMaps that already exist in your app. Unlike Account migrations, CoMap migrations are not run when a CoMap is created.

Note: Migrations are run synchronously and cannot be run asynchronously.

Here's an example of a migration that adds the `priority` field to the `Task` CoMap:

```
const Task = co
  .map({
    done: z.boolean(),
    text: co.plainText(),
    version: z.literal([1, 2]),
    priority: z.enum(["low", "medium", "high"]), // new field
  })
  .withMigration((task) => {
    if (task.version === 1) {
      task.$jazz.set("priority", "medium");
      // Upgrade the version so the migration won't run again
      task.$jazz.set("version", 2);
    }
  });
```

Copy

Migration best practices

Design your schema changes to be compatible with existing data:

- **Add, don't change:** Only add new fields; avoid renaming or changing types of existing fields
- **Make new fields optional:** This prevents errors when loading older data
- **Use version fields:** Track schema versions to run migrations only when needed

Migration & reader permissions

Migrations need write access to modify CoMaps. If some users only have read permissions, they can't run migrations on those CoMaps.

Forward-compatible schemas (where new fields are optional) handle this gracefully - users can still use the app even if migrations haven't run.

Non-compatible changes require ha in your app code using discriminated unions.

When you can't guarantee all users can run migrations, handle multiple schema versions explicitly:

```
const TaskV1 = co.map({
  version: z.literal(1),
  done: z.boolean(),
  text: z.string(),
});

const TaskV2 = co
  .map({
    // We need to be more strict about the version to make the
    // discriminated union work
    version: z.literal(2),
    done: z.boolean(),
    text: z.string(),
    priority: z.enum(["low", "medium", "high"]),
  })
  .withMigration((task) => {
    if (task.version === 1) {
      task.$jazz.set("version", 2);
      task.$jazz.set("priority", "medium");
    }
  });

// Export the discriminated union; because some users might
// not be able to run the migration
export const Task = co.discriminatedUnion("version", [TaskV1, TaskV2]);
export type Task = co.loaded<typeof Task>;
```

Copy

Best Practices

Structuring Data

- Use struct-like CoMaps for entities with fixed, known properties
- Use record-like CoMaps for dynamic key-value collections
- Group related properties into nested CoMaps for better organization

Common Patterns

Helper methods

You should define helper methods of CoValue schemas separately, in standalone functions:

```
import { co, z } from "jazz-tools";

const Project = co.map({
  name: z.string(),
```

Copy

```
  startDate: z.date(),
  endDate: z.optional(z.date()),
});
type Project = co.loaded<typeof Project>;

export function isProjectActive(project: Project) {
  const now = new Date();
  return (
    now >= project.startDate && (!project.endDate || now <= project.endDate)
  );
}

export function formatProjectDuration(
  project: Project,
  format: "short" | "full",
) {
  const start = project.startDate.toLocaleDateString();
  if (!project.endDate) {
    return format === "full" ? `Started on ${start}, ongoing` : `From ${start}`;
  }

  const end = project.endDate.toLocaleDateString();
  return format === "full"
    ? `From ${start} to ${end}`
    : `${(project.endDate.getTime() - project.startDate.getTime()) / 86400000} days`;
}

const project = Project.create({
  name: "My project",
  startDate: new Date("2025-04-01"),
  endDate: new Date("2025-04-04"),
});

console.log(isProjectActive(project)); // false
console.log(formatProjectDuration(project, "short")); // "3 days"
```

Uniqueness

CoMaps are typically created with a CoValue ID that acts as an opaque UUID, by which you can then load them. However, there are situations where it is preferable to load CoMaps using a custom identifier:

- The CoMaps have user-generated identifiers, such as a slug
- The CoMaps have identifiers referring to equivalent data in an external system
- The CoMaps have human-readable & application-specific identifiers
  - If an application has CoValues used by every user, referring to it by a unique *well-known* name (eg, `"my-global-coma` han using a CoValue ID

Consider a scenario where one wants to identify a CoMap using some unique identifier that isn't the Jazz CoValue ID:

```
// This will not work as `learning-jazz` is not a CoValue ID
const myTask = await Task.load("learning-jazz");
```

Copy

To make it possible to use human-readable identifiers Jazz lets you to define a `unique` property on CoMaps.

Then the CoValue ID is deterministically derived from the `unique` property and the owner of the CoMap.

```
// Given the project owner, myTask will have always the same id
Task.create(
  {
    text: "Let's learn some Jazz!",
  },
  {
    unique: "learning-jazz",
    owner: project.$jazz.owner, // Different owner, different id
  },
);
```

Copy

Now you can use `CoMap.loadUnique` to easily load the CoMap using the human-readable identifier:

```
const learnJazzTask = await Task.loadUnique(
  "learning-jazz",
  project.$jazz.owner.$jazz.id,
);
```

Copy

It's also possible to combine the create+load operation using `CoMap.upsertUnique`:

```
await Task.upsertUnique({
  value: {
    text: "Let's learn some Jazz!",
  },
  unique: "learning-jazz",
  owner: project.$jazz.owner,
});
```

Copy

Caveats:

- The `unique` parameter acts as an *immutable* identifier - i.e. the same `unique` parameter in the same `Group` will always refer to the same CoValue.
  - To make dynamic renaming possible, you can create an indirection where a stable CoMap identified by a specific valu[illegible]r to another CoMap with a normal, dynamic CoValue ID. This pointer can then be updated as desired by users with the

corresponding permissions.

- This way of introducing identifiers allows for very fast lookup of individual CoMaps by identifier, but it doesn't let you enumerate all the CoMaps identified this way within a `Group`. If you also need enumeration, consider using a global `co.record()` that maps from identifier to a CoMap, which you then do lookups in (this requires at least a shallow load of the entire `co.record()`, but this should be fast for up to 10s of 1000s of entries)

Creating Set-like Collections

You can use CoRecords as a way to create set-like collections, by keying the CoRecord on the item's CoValue ID. You can then use static `Object` methods to iterate over the CoRecord, effectively allowing you to treat it as a set.

```
const Chat = co.map({
  messages: co.list(Message),
  participants: co.record(z.string(), MyAppUser),
});

const chat = await Chat.load(chatId, {
  resolve: {
    participants: true,
  },
});

let participantList: string[];

// Note that I don't need to load the map deeply to read and set keys
if (chat.$isLoaded) {
  chat.participants.$jazz.set(me.$jazz.id, me);
  participantList = Object.keys(chat.participants);
}
```

Copy

You can choose a loading strategy for the CoRecord. Use $each when you need all item properties to be immediately available. In general, it is enough to shallowly load a CoRecord to access its keys, and then load the values of those keys as needed (for example, by passing the keys as strings to a child component).

```
const { participants } = await chat.$jazz.ensureLoaded({
  resolve: {
    participants: {
      $each: {
        profile: {
          avatar: true,
        },
      },
    },
  },
});
```

Copy

```
const avatarList = Object.values(participants).map(
  (user) => user.profile.avatar,
);
```

Was this page helpful? Yes No

Docs issue? Join Discord

garden computing

Computers are magic.
Time to make them less complex.

Stay up to date

Enter your email Subscribe

About

Status

Team

Blog

Resources

Documentation

Examples

Pricing

Showcase

CoLists

CoLists are ordered collections that work like JavaScript arrays. They provide indexed access, iteration methods, and length properties, making them perfect for managing sequences of items.

Creating CoLists

CoLists are defined by specifying the type of items they contain:

```
import { co, z } from "jazz-tools";

const ListOfResources = co.list(z.string());
export type ListOfResources = co.loaded<typeof ListOfResources>;

const ListOfTasks = co.list(Task);
export type ListOfTasks = co.loaded<typeof ListOfTasks>;
export type ListOfTasksInitShape = co.input<typeof ListOfTasks>; // type accepted by
`ListOfTasks.create`
```

To create a `CoList`:

```
// Create an empty list
const resources = co.list(z.string()).create([]);

// Create a list with initial items
const tasks = co.list(Task).create([
  { title: "Prepare soil beds", status: "in-progress" },
  { title: "Order compost", status: "todo" },
]);
```

Ownership

Like other CoValues, you can specify ownership when creating CoLists.

```
// Create with shared ownership
const teamGroup = Group.create();
teamGroup.addMember(colleagueAccount, "writer");

const teamList = co.list(Task).create([], { owner: teamGroup });
```

See Groups as permission scopes fo use groups to control access to CoLists.

Reading from CoLists

CoLists support standard array access patterns:

```
// Access by index
const firstTask = tasks[0];
console.log(firstTask.title); // "Prepare soil beds"

// Get list length
console.log(tasks.length); // 2

// Iteration
tasks.forEach((task) => {
  console.log(task.title);
  // "Prepare soil beds"
  // "Order compost"
});

// Array methods
const todoTasks = tasks.filter((task) => task.status === "todo");
console.log(todoTasks.length); // 1
```

Copy

Updating CoLists

Methods to update a CoList's items are grouped inside the `$jazz` namespace:

```
// Add items
resources.$jazz.push("Tomatoes"); // Add to end
resources.$jazz.unshift("Lettuce"); // Add to beginning
tasks.$jazz.push({
  // Add complex items
  title: "Install irrigation", // (Jazz will create
  status: "todo", // the CoValue for you!)
});

// Replace items
resources.$jazz.set(0, "Cucumber"); // Replace by index

// Modify nested items
tasks[0].$jazz.set("status", "complete"); // Update properties of references
```

Copy

Soft Deletion

You can do a soft deletion by using a deleted flag, then creating a helper method that explicitly filters out items where the deleted property

Copy

```
const Task = co.map({
  title: z.string(),
  status: z.literal(["todo", "in-progress", "complete"]),
  deleted: z.optional(z.boolean()),
});
type Task = typeof Task;

const ListOfTasks = co.list(Task);
type ListOfTasks = typeof ListOfTasks;

export function getCurrentTasks(list: co.loaded<ListOfTasks, { $each: true }>) {
  return list.filter((task): task is co.loaded<Task> => !task.deleted);
}

async function main() {
  const myTaskList = ListOfTasks.create([]);
  myTaskList.$jazz.push({
    title: "Tomatoes",
    status: "todo",
    deleted: false,
  });
  myTaskList.$jazz.push({
    title: "Cucumbers",
    status: "todo",
    deleted: true,
  });
  myTaskList.$jazz.push({
    title: "Carrots",
    status: "todo",
  });

  const activeTasks = getCurrentTasks(myTaskList);
  console.log(activeTasks.map((task) => task.title));
  // Output: ["Tomatoes", "Carr
}
```

There are several benefits to soft deletions:

- **recoverablity** - Nothing is truly deleted, so recovery is possible in the future
- **data integrity** - Relationships can be maintained between current and deleted values
- **auditable** - The data can still be accessed, good for audit trails and checking compliance

Deleting Items

Jazz provides two methods to retain or remove items from a CoList:

```
// Remove items
resources.$jazz.remove(2); // By index
console.log(resources); // ["Cucumber", "Peppers"]
resources.$jazz.remove((item) => item === "Cucumber"); // Or by predicate
console.log(resources); // ["Tomatoes", "Peppers"]

// Keep only items matching the predicate
resources.$jazz.retain((item) => item !== "Cucumber");
console.log(resources); // ["Tomatoes", "Peppers"]
```

Copy

You can also remove specific items by index with `splice`, or remove the first or last item with `pop` or `shift`:

```
// Remove 2 items starting at index 1
resources.$jazz.splice(1, 2);
console.log(resources); // ["Tomatoes"]

// Remove a single item at index 0
resources.$jazz.splice(0, 1);
console.log(resources); // ["Cucumber", "Peppers"]

// Remove items
const lastItem = resources.$jazz.pop(); // Remove and return last item
resources.$jazz.shift(); // Remove first item
```

Copy

Array Methods

`CoList`s support the standard JavaScript array methods you already know. Methods that mutate the array are grouped inside the `$jazz` namespace.

```
// Add multiple items at once
resources.$jazz.push("Tomatoes", "Basil", "Peppers");

// Find items
const basil = resources.find((r) => r === "Basil");

// Filter (returns regular array, not a CoList)
```

Copy

```
const tItems = resources.filter((r) => r.startsWith("T"));
console.log(tItems); // ["Tomatoes"]
```

Type Safety

CoLists maintain type safety for their items:

```
// TypeScript catches type errors
resources.$jazz.push("Carrots"); // ✓ Valid string
resources.$jazz.push(42); // ✗ Type error: expected string
// Argument of type 'number' is not assignable to parameter of type 'string'
// For lists of references
tasks.forEach((task) => {
  console.log(task.title); // TypeScript knows task has title
});
```

Copy

Best Practices

Common Patterns

List Rendering

CoLists work well with UI rendering libraries:

```
import { co, z } from "jazz-tools";
const ListOfTasks = co.list(Task);

// React example
function TaskList({ tasks }: { tasks: co.loaded<typeof ListOfTasks> }) {
  return (
    <ul>
      {tasks.map((task) =>
        task.$isLoaded ? (
          <li key={task.$jazz.id}>
            {task.title} - {task.status}
          </li>
        ) : null,
      )}
    </ul>
  );
}
```

Copy

Managing Relations

CoLists can be used to create one-to-many relationships:

```
import { co, z } from "jazz-to
```

Copy

```ts
const Task = co.map({
  title: z.string(),
  status: z.literal(["todo", "in-progress", "complete"]),

  get project(): co.Optional<typeof Project> {
    return co.optional(Project);
  },
});

const ListOfTasks = co.list(Task);

const Project = co.map({
  name: z.string(),

  get tasks(): co.List<typeof Task> {
    return ListOfTasks;
  },
});

const project = Project.create({
  name: "Garden Project",
  tasks: ListOfTasks.create([]),
});

const task = Task.create({
  title: "Plant seedlings",
  status: "todo",
  project: project, // Add a reference to the project
});

// Add a task to a garden project
project.tasks.$jazz.push(task);

// Access the project from the task
console.log(task.project); // { name: "Garden Project", tasks: [task] }
```

Set-like Collections

CoLists, like JavaScript arrays, allow you to insert the same item multiple times. In some cases, you might want to have a collection of unique items (similar to a set). To achieve this, you can use a CoRecord with entries keyed on a unique identifier (for example, the CoValue ID).

You can read more about this pattern here.

garden computing

Computers are magic.
Time to make them less complex.

Stay up to date

Enter your email

Subscribe

About

Status

Team

Blog

Resources

Documentation

Examples

Pricing

Showcase