Model

Base

View source

Model class representing a base.

If you want the base model to automatically recalculate whenever the base schema changes, try the useBase hook.

Members

`class Base extends BaseCore<InterfaceSdkMode>`

`readonly activeCollaborators`

`Array<CollaboratorData>`

The users who have access to this base.

```
import {useBase} from '@airtable/blocks/interface/ui';

function MyApp() {
    const base = useBase();
    console.log(base.activeCollaborators[0].email);
}
```

`readonly color`

`string`

The color of the base.

```
import {colorUtils, useBase} from '@airtable/blocks/interface/ui';

function MyApp() {
    const base = useBase();
    return (
        <div style={{backgroundColor: colorUtils.getHexForColor(base.color)}}>
            This div's background is the same color as the base background
        </div>
    );
}
```

`readonly id`

`string`

The ID for this model.

`readonly`

`boolean`

`isDeleted`

`true` if the model has been deleted, and `false` otherwise.

In general, it's best to avoid keeping a reference to an object past the current event loop, since it may be deleted and trying to access any data of a deleted object (other than its ID) will throw. But if you keep a reference, you can use `isDeleted` to check that it's safe to access the model's data.

`readonly name`

`string`

The name of the base.

```
import {useBase} from '@airtable/blocks/interface/ui';

function MyApp() {
    const base = useBase();
    console.log('The name of your base is', base.name);
}
```

`readonly tables`

`Array<Table>`

The tables in this base. Can be watched to know when tables are created, deleted, or reordered in the base.

```
import {useBase} from '@airtable/blocks/interface/ui';

function MyApp() {
    const base = useBase();
    console.log(`You have ${base.tables.length} tables`);
}
```

`readonly workspaceId`

`string`

The workspace id of the base.

```
import {useBase} from '@airtable/blocks/interface/ui';

function MyApp() {
    const base = useBase();
    console.log('The workspace id of your base is', base.workspaceId);
}
```

`getCollaborator`

`function (idOrNameOrEmail: UserId | string) => CollaboratorData | null`

	The user matching the given ID, name, or email address. Throws if that user does not exist or does not have access to this base. Use getCollaboratorIfExists instead if you are unsure whether a collaborator with the given ID exists and has access to this base. This method is convenient when building an extension for a specific base, but for more generic extensions the best practice is to use the getCollaboratorById method instead.
`getCollaboratorById`	`function (collaboratorId: UserId) => CollaboratorData` `collaboratorId` The ID of the user. The user matching the given ID. Throws if that user does not exist or does not have access to this base. Use getCollaboratorByIdIfExists instead if you are unsure whether a collaborator with the given ID exists and has access to this base.
`getCollaboratorByIdIfExists`	`function (collaboratorId: UserId) => CollaboratorData | null` `collaboratorId` The ID of the user. The user matching the given ID, or `null` if that user does not exist or does not have access to this base.
`getCollaboratorIfExists`	`function (idOrNameOrEmail: UserId | string) => CollaboratorData | null` The user matching the given ID, name, or email address. Returns null if that user does not exist or does not have access to this base. This method is convenient when building an extension for a specific base, but for more generic extensions the best practice is to use the getCollaboratorByIdIfExists method instead.
`getMaxRecordsPerTable`	`function () => number` Returns the maximum number of records allowed in each table of this base.
`getTable`	`function (tableIdOrName: TableId | string) => Table` `tableIdOrName` The ID or name of the table you're looking for. The table matching the given ID or name. Throws if no matching table exists within this base. Use getTableIfExists instead if you are unsure whether a table exists with the given

	name/ID. This method is convenient when building an extension for a specific base, but for more generic extensions the best practice is to use the getTableById or getTableByName methods instead.
`getTableById`	`function (tableId: string) => Table` `tableId` The ID of the table. The table matching the given ID. Throws if that table does not exist in this base. Use getTableByIdIfExists instead if you are unsure whether a table exists with the given ID.
`getTableByIdIfExists`	`function (tableId: string) => Table | null` `tableId` The ID of the table. The table matching the given ID, or `null` if that table does not exist in this base.
`getTableByName`	`function (tableName: string) => Table` `tableName` The name of the table you're looking for. The table matching the given name. Throws if no table exists with that name in this base. Use getTableByNameIfExists instead if you are unsure whether a table exists with the given name.
`getTableByNameIfExists`	`function (tableName: string) => Table | null` `tableName` The name of the table you're looking for. The table matching the given name, or `null` if no table exists with that name in this base.
`getTableIfExists`	`function (tableIdOrName: TableId | string) => Table | null` `tableIdOrName` The ID or name of the table you're looking for. The table matching the given ID or name. Returns `null` if no matching table exists within this base. This method is convenient when building an extension for a specific base, but for more generic extensions the best practice is to use the getTableByIdIfExists or getTableByNameIfExists methods instead.
`toString`	`function () => string`

A string representation of the model for use in debugging.

`unwatch`	`function (keys: WatchableBaseKey	ReadonlyArray<WatchableBaseKey>, callback: function (model: this, key: WatchableBaseKey, args: ...Array<any>) => unknown, context?: FlowAnyObject	null) => Array<WatchableBaseKey>` `keys` — the keys to unwatch `callback` — the function passed to `.watch` for these keys `context` — the context that was passed to `.watch` for this `callback` Unwatch keys watched with `.watch`. Should be called with the same arguments given to `.watch`. Returns the array of keys that were unwatched.
`watch`	`function (keys: WatchableBaseKey	ReadonlyArray<WatchableBaseKey>, callback: function (model: this, key: WatchableBaseKey, args: ...Array<any>) => unknown, context?: FlowAnyObject	null) => Array<WatchableBaseKey>` `keys` — the keys to watch `callback` — a function to call when those keys change `context` — an optional context for `this` in `callback`. Get notified of changes to the model. Every call to `.watch` should have a matching call to `.unwatch`. Returns the array of keys that were watched.

Model

Field

View source

Model class representing a field in a table.

```
import {useBase} from '@airtable/blocks/interface/ui';

function App() {
    const base = useBase();
    const table = base.getTableByName('Table 1');
    const field = table.getFieldByName('Name');
    console.log('The type of this field is', field.type);
}
```

Members

`class Field extends FieldCore<InterfaceSdkMode>`

`readonly config`

`FieldConfig`

The type and options of the field to make type narrowing `FieldOptions` easier. See FieldType for r
information on the options for each field type.

```
const fieldConfig = field.config;
if (fieldConfig.type === FieldType.SINGLE_SELECT) {
    return fieldConfig.options.choices;
} else if (fieldConfig.type === FieldType.MULTIPLE_LOOKUP_VALUES && fieldConfig.options.isV
    if (fieldConfig.options.result.type === FieldType.SINGLE_SELECT) {
        return fieldConfig.options.result.options.choices;
    }
}
return DEFAULT_CHOICES;
```

`readonly description`

`string | null`

The description of the field, if it has one. Can be watched.

```
console.log(myField.description);
// => 'This is my field'
```

`readonly id`

`string`

The ID for this model.

`readonly isComputed`

`boolean`

`true` if this field is computed, `false` otherwise. A field is "computed" if it's value is not set by user input (e.g. autoNumber, formula, etc.). Can be watched

```
console.log(mySingleLineTextField.isComputed);
// => false
console.log(myAutoNumberField.isComputed);
// => true
```

`readonly isDeleted`

`boolean`

`true` if the model has been deleted, and `false` otherwise.

In general, it's best to avoid keeping a reference to an object past the current event loop, since it may be deleted and trying to access any data of a deleted object (other than its ID) will throw. But if you keep a reference, you can use `isDeleted` to check that it's safe to access the model's data.

`readonly isPrimaryField`

`boolean`

`true` if this field is its parent table's primary field, `false` otherwise. Should never change because the primary field of a table cannot change.

`readonly name`

`string`

The name of the field. Can be watched.

```
console.log(myField.name);
// => 'Name'
```

`readonly options`

`FieldOptions | null`

The configuration options of the field. The structure of the field's options depend on the field's type. `null` if the field has no options. See FieldType for more information on the options for each field type. Can be watched.

```
import {FieldType} from '@airtable/blocks/interface/models';

if (myField.type === FieldType.CURRENCY) {
    console.log(myField.options.symbol);
    // => '$'
}
```

`readonly type FieldType`

The type of the field. Can be watched.

```
console.log(myField.type);
// => 'singleLineText'
```

`convertStringToCellValue`

`function (string: string) => unknown`

`string`	The string to parse.

Attempt to parse a given string and return a valid cell value for the field's current config. Returns `null` if unable to parse the given string.

```
const inputString = '42';
const cellValue = myNumberField.convertStringToCellValue(inputString);
console.log(cellValue === 42);
// => true
```

`toString`

`function () => string`

A string representation of the model for use in debugging.

`unwatch`

`function (keys: WatchableFieldKey | ReadonlyArray<WatchableFieldKey>, callback: function (model: this, key: WatchableFieldKey, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableFieldKey>`

`keys`	the keys to unwatch
`callback`	the function passed to `.watch` for these keys
`context`	the context that was passed to `.watch` for this `callback`

Unwatch keys watched with `.watch`.

Should be called with the same arguments given to `.watch`.

Returns the array of keys that were unwatched.

`watch`

`function (keys: WatchableFieldKey | ReadonlyArray<WatchableFieldKey>, callback: function (model: this, key: WatchableFieldKey, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableFieldKey>`

`keys`	the keys to watch

`callback`	a function to call when those keys change
`context`	an optional context for `this` in `callback`.

Get notified of changes to the model.

Every call to `.watch` should have a matching call to `.unwatch`.

Returns the array of keys that were watched.

GlobalConfig

View source

A key-value store for persisting configuration options for an extension installation.

The contents will be synced in real-time to all logged-in users of the installation. Contents will not be updated in real-time when the installation is running in a publicly shared base.

Any key can be watched to know when the value of the key changes. If you want your component to automatically re-render whenever any key on GlobalConfig changes, try using the useGlobalConfig hook.

You should not need to construct this object yourself.

The maximum allowed size for a given GlobalConfig instance is 150kB. The maximum number of keys for a given GlobalConfig instance is 1000.

Members

`class GlobalConfig extends Watchable<WatchableGlobalConfigKey>`

checkPermissionsForSet

`function (key?: PartialGlobalConfigKey, value?: GlobalConfigValue) => PermissionCheckResult`

`key`	A string for the top-level key, or an array of strings describing the path to set.
`value`	The value to set at the specified path. Use `undefined` to delete the value at the given path.

Checks whether the current user has permission to set the given global config key.

Accepts partial input, in the same format as setAsync. The more information provided, the more accurate the permissions check will be.

Returns `{hasPermission: true}` if the current user can set the specified key, `{hasPermission: false, reasonDisplayString: string}` otherwise. `reasonDisplayString` may be used to display an error message to the user.

```
// Check if user can update a specific key and value.
const setCheckResult =
    globalConfig.checkPermissionsForSet('favoriteColor', 'purple');
if (!setCheckResult.hasPermission) {
    alert(setCheckResult.reasonDisplayString);
}

// Check if user can update a specific key without knowing the value
const setKeyCheckResult =
    globalConfig.checkPermissionsForSet('favoriteColor');

// Check if user can update globalConfig without knowing key or value
const setUnknownKeyCheckResult = globalConfig.checkPermissionsForSet();
```

checkPermissionsForSetPaths

`function (updates?: ReadonlyArray<PartialGlobalConfigUpdate>) => PermissionCheckResult`

`updates` The paths and values to set.

Checks whether the current user has permission to perform the specified updates to global config.

Accepts partial input, in the same format as setPathsAsync. The more information provided, the more accurate the permissions check will be.

Returns `{hasPermission: true}` if the current user can set the specified key, `{hasPermission: false, reasonDisplayString: string}` otherwise. `reasonDisplayString` may be used to display an error message to the user.

```
// Check if user can update a specific keys and values.
const setPathsCheckResult = globalConfig.checkPermissionsForSet([
    {path: ['topLevelKey1', 'nestedKey1'], value: 'foo'},
    {path: ['topLevelKey2', 'nestedKey2'], value: 'bar'},
]);
if (!setPathsCheckResult.hasPermission) {
    alert(setPathsCheckResult.reasonDisplayString);
}

// Check if user could potentially set globalConfig values.
// Equivalent to globalConfig.checkPermissionsForSet()
const setUnknownPathsCheckResult =
    globalConfig.checkPermissionsForSetPaths();
```

get

`function (key: GlobalConfigKey) => unknown`

`key` A string for the top-level key, or an array of strings describing the path to the value.

Get the value at a path. Throws an error if the path is invalid.

Returns undefined if no value exists at that path.

```
import {useGlobalConfig} from '@airtable/blocks/interface/ui';

function MyApp() {
    const globalConfig = useGlobalConfig();
    const topLevelValue = globalConfig.get('topLevelKey');
    const nestedValue = globalConfig.get(['topLevelKey', 'nested', 'deeply']);
}
```

`hasPermissionToSet`

`function (key?: PartialGlobalConfigKey, value?: GlobalConfigValue) => boolean`

`key`	A string for the top-level key, or an array of strings describing the path to set.
`value`	The value to set at the specified path. Use `undefined` to delete the value at the given path.

An alias for `globalConfig.checkPermissionsForSet(key, value).hasPermission`.

Checks whether the current user has permission to set the given global config key.

Accepts partial input, in the same format as setAsync. The more information provided, the more accurate the permissions check will be.

```
// Check if user can update a specific key and value.
const canSetFavoriteColorToPurple =
    globalConfig.hasPermissionToSet('favoriteColor', 'purple');
if (!canSetFavoriteColorToPurple) {
    alert('Not allowed!');
}

// Check if user can update a specific key without knowing the value
const canSetFavoriteColor = globalConfig.hasPermissionToSet('favoriteColor');

// Check if user can update globalConfig without knowing key or value
const canSetGlobalConfig = globalConfig.hasPermissionToSet();
```

`hasPermissionToSetPaths`

`function (updates?: ReadonlyArray<PartialGlobalConfigUpdate>) => boolean`

`updates`	The paths and values to set.

An alias for `globalConfig.checkPermissionsForSetPaths(updates).hasPermission`.

Checks whether the current user has permission to perform the specified updates to global config.

Accepts partial input, in the same format as setPathsAsync. The more information provided, the more accurate the permissions check will be.

```
// Check if user can update a specific keys and values.
const canSetPaths = globalConfig.hasPermissionToSetPaths([
    {path: ['topLevelKey1', 'nestedKey1'], value: 'foo'},
    {path: ['topLevelKey2', 'nestedKey2'], value: 'bar'},
]);
if (!canSetPaths) {
    alert('not allowed!');
}

// Check if user could potentially set globalConfig values.
// Equivalent to globalConfig.hasPermissionToSet()
const canSetAnyPaths = globalConfig.hasPermissionToSetPaths();
```

setAsync

`function (key: GlobalConfigKey, value?: GlobalConfigValue) => Promise<void>`

`key`	A string for the top-level key, or an array of strings describing the path to set.
`value`	The value to set at the specified path. Use `undefined` to delete the value at the given path.

Sets a value at a path. Throws an error if the path or value is invalid.

This action is asynchronous: `await` the returned promise if you wish to wait for the update to be persisted to Airtable servers.

Updates are applied optimistically locally, so your change will be reflected in GlobalConfig before the promise resolves.

```
import {useGlobalConfig} from '@airtable/blocks/interface/ui';

function MyApp() {
    const globalConfig = useGlobalConfig();
    const updateFavoriteColorIfPossible = (color) => {
        if (globalConfig.hasPermissionToSet('favoriteColor', color)) {
            globalConfig.setAsync('favoriteColor', color);
        }
        // The update is now applied within your extension (eg will be
        // reflected in globalConfig) but are still being saved to
        // Airtable servers (e.g. may not be updated for other users yet)
    }

    const updateFavoriteColorIfPossibleAsync = async (color) => {
        if (globalConfig.hasPermissionToSet('favoriteColor', color)) {
            await globalConfig.setAsync('favoriteColor', color);
        }
        // globalConfig updates have been saved to Airtable servers.
        alert('favoriteColor has been updated');
    }
}
```

setPathsAsync

`function (updates: Array<GlobalConfigUpdate>) => Promise<void>`

`updates` The paths and values to set.

Sets multiple values. Throws if any path or value is invalid.

This action is asynchronous: `await` the returned promise if you wish to wait for the updates t
be persisted to Airtable servers. Updates are applied optimistically locally, so your changes wil
be reflected in GlobalConfig before the promise resolves.

```
import {useGlobalConfig} from '@airtable/blocks/interface/ui';

function MyApp() {
    const globalConfig = useGlobalConfig();
    const updates = [
        {path: ['topLevelKey1', 'nestedKey1'], value: 'foo'},
        {path: ['topLevelKey2', 'nestedKey2'], value: 'bar'},
    ];

    const applyUpdatesIfPossible = () => {
        if (globalConfig.hasPermissionToSetPaths(updates)) {
            globalConfig.setPathsAsync(updates);
        }
        // The updates are now applied within your extension (eg will be reflected
        // globalConfig) but are still being saved to Airtable servers (e.g. they
        // may not be updated for other users yet)
    }

    const applyUpdatesIfPossibleAsync = async () => {
        if (globalConfig.hasPermissionToSetPaths(updates)) {
            await globalConfig.setPathsAsync(updates);
        }
        // globalConfig updates have been saved to Airtable servers.
        alert('globalConfig has been updated');
    }
}
```

unwatch

`function (keys: WatchableGlobalConfigKey | ReadonlyArray<WatchableGlobalConfigKey>, callback: function (model: this, key: WatchableGlobalConfigKey, args: ...Array<any>) => unknown, context?: FlowAnyObject |`

`null) => Array<WatchableGlobalConfigKey>`

`keys`	the keys to unwatch
`callback`	the function passed to `.watch` for these keys
`context`	the context that was passed to `.watch` for this `callback`

Unwatch keys watched with `.watch`.

Should be called with the same arguments given to `.watch`.

Returns the array of keys that were unwatched.

`watch`

```
function (keys: WatchableGlobalConfigKey |
ReadonlyArray<WatchableGlobalConfigKey>, callback: function
(model: this, key: WatchableGlobalConfigKey,
args: ...Array<any>) => unknown, context?: FlowAnyObject |
null) => Array<WatchableGlobalConfigKey>
```

`keys`	the keys to watch
`callback`	a function to call when those keys change
`context`	an optional context for `this` in `callback`.

Get notified of changes to the model.

Every call to `.watch` should have a matching call to `.unwatch`.

Returns the array of keys that were watched.

Model

Record

View source

Model class representing a record in a table.

Do not instantiate. You can get instances of this class by calling useRecords.

Members

`class Record extends RecordCore<InterfaceSdkMode, WatchableRecordKey>`

`readonly createdTime`

`Date`

The created time of this record.

```
console.log(`
    This record was created at ${myRecord.createdTime.toISOString()}
`);
```

`readonly id`

`string`

The ID for this model.

`readonly isDeleted`

`boolean`

`true` if the model has been deleted, and `false` otherwise.

In general, it's best to avoid keeping a reference to an object past the current event loop, since it may be deleted and trying to access any data of a deleted object (other than its ID) will throw. But if you keep a reference, you can use `isDeleted` to check that it's safe to access the model's data.

`readonly name`

`string`

The primary cell value in this record, formatted as a `string`.

```
console.log(myRecord.name);
// => '42'
```

`fetchForeignRecordsAsync`

```
function (field: Field, filterString:
string) => Promise<{
records: ReadonlyArray<{
```

```
id: RecordId;
name: string;
}>;
}>
```

`filterString`	The filter string to use to filter the records.

Fetch foreign records for a field. Subsequent calls to this method will override previous calls that are still pending. The previous call(s) will immediately resolve with an empty `records` array.

`getCellValue`	`function (fieldOrFieldIdOrFieldName: Field | FieldId | string) => unknown` `fieldOrFieldIdOrFieldName` — The field (or field ID or field name) whose cell value you'd like to get. Gets the cell value of the given field for this record.

```
const cellValue = myRecord.getCellValue(mySingleLineTextField);
console.log(cellValue);
// => 'cell value'
```

`getCellValueAsString`	`function (fieldOrFieldIdOrFieldName: Field | FieldId | string) => string` `fieldOrFieldIdOrFieldName` — The field (or field ID or field name) whose cell value you'd like to get. Gets the cell value of the given field for this record, formatted as a `string`.

```
const stringValue = myRecord.getCellValueAsString(myNumberField);
console.log(stringValue);
// => '42'
```

`toString`	`function () => string` A string representation of the model for use in debugging.
`unwatch`	`function (keys: WatchableRecordKeyCore | WatchableRecordKey | ReadonlyArray<WatchableRecordKeyCore | WatchableRecordKey>, callback: function (model: this, key: WatchableRecordKeyCore | WatchableRecordKey, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableRecordKeyCore | WatchableRecordKey>`  `keys` — the keys to unwatch

`callback`	the function passed to `.watch` for these keys
`context`	the context that was passed to `.watch` for this `callback`

Unwatch keys watched with `.watch`.

Should be called with the same arguments given to `.watch`.

Returns the array of keys that were unwatched.

`watch`

```
function (keys: WatchableRecordKeyCore | WatchableRecordKey |
ReadonlyArray<WatchableRecordKeyCore | WatchableRecordKey>,
callback: function (model: this, key: WatchableRecordKeyCore |
WatchableRecordKey, args: ...Array<any>) => unknown, context?:
FlowAnyObject | null) => Array<WatchableRecordKeyCore |
WatchableRecordKey>
```

`keys`	the keys to watch
`callback`	a function to call when those keys change
`context`	an optional context for `this` in `callback`.

Get notified of changes to the model.

Every call to `.watch` should have a matching call to `.unwatch`.

Returns the array of keys that were watched.

Model

Session

View source

Model class representing the current user's session.

```
import {useSession} from '@airtable/blocks/interface/ui';

function Username() {
    const session = useSession();

    if (session.currentUser !== null) {
        return <span>The current user's name is {session.currentUser.name}</span>;
    } else {
        return <span>This extension is being viewed in a public share</span>;
    }
}
```

Members

`class Session extends SessionCore<InterfaceSdkMode>`

`readonly currentUser`	`CollaboratorData | null`
	
	The current user, or `null` if the extension is running in a publicly shared base.
	
	`import {useSession} from '@airtable/blocks/interface/ui';`
	
	`function CurrentUser() {`
	`    const session = useSession();`
	
	`    if (!session.currentUser) {`
	`        return <div>This extension is being used in a public share.</div>;`
	`    }`
	
	`    return <ul>`
	`        <li>ID: {session.currentUser.id}</li>`
	`        <li>E-mail: {session.currentUser.email}</li>`
	`        <li>Name: {session.currentUser.name}</li>`
	`    </ul>;`
	`}`

`readonly id`	`string`

	The ID for this model.
`readonly isDeleted`	`boolean` `true` if the model has been deleted, and `false` otherwise. In general, it's best to avoid keeping a reference to an object past the current event loop, since it may be deleted and trying to access any data of a deleted object (other than its ID) will throw. But if you keep a reference, you can use `isDeleted` to check that it's safe to access the model's data.
`toString`	`function () => string` A string representation of the model for use in debugging.

`unwatch`	`function (keys: WatchableSessionKey | ReadonlyArray<WatchableSessionKey>, callback: function (model: this, key: WatchableSessionKey, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableSessionKey>`
	
	`keys` the keys to unwatch
	
	`callback` the function passed to `.watch` for these keys
	
	`context` the context that was passed to `.watch` for this `callback`
	
	Unwatch keys watched with `.watch`.
	
	Should be called with the same arguments given to `.watch`.
	
	Returns the array of keys that were unwatched.

`watch`	`function (keys: WatchableSessionKey | ReadonlyArray<WatchableSessionKey>, callback: function (model: this, key: WatchableSessionKey, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableSessionKey>` `keys` the keys to watch `callback` a function to call when those keys change `context` an optional context for `this` in `callback`.

Get notified of changes to the model.

Every call to `.watch` should have a matching call to `.unwatch`.

Returns the array of keys that were watched.

Table

View source

Model class representing a table. Every Base has one or more tables.

```
import {useBase} from '@airtable/blocks/interface/ui';

function App() {
    const base = useBase();
    const table = base.getTables()[0];
    if (table) {
        console.log('The name of this table is', table.name);
    }
}
```

Members

`class Table extends TableCore<InterfaceSdkMode>`

`readonly description`	`string | null`
	
	The description of the table, if it has one. Can be watched.
	
	`console.log(myTable.description);`
	`// => 'This is my table'`

`readonly fields`	`Array<Field>`
	
	The fields in this table. The order is arbitrary, since fields are only ordered in the context of a specific view.
	
	Can be watched to know when fields are created or deleted.
	
	``console.log(`This table has ${myTable.fields.length} fields`);``

`readonly id`	`string` The ID for this model.
`readonly isDeleted`	`boolean` `true` if the model has been deleted, and `false` otherwise.

In general, it's best to avoid keeping a reference to an object past the current event loop, since it may be deleted and trying to access any data of a deleted object (other than its ID) will throw. But if you keep a reference, you can use `isDeleted` to check that it's safe to access the model's data.

`readonly name`

`string`

The name of the table. Can be watched.

```
console.log(myTable.name);
// => 'Table 1'
```

`readonly primaryField`

[`Field`]

The table's primary field. Every table has exactly one primary field. The primary field of a table will not change.

```
console.log(myTable.primaryField.name);
// => 'Name'
```

`checkPermissionToExpandRecords`

`function () =>` [`PermissionCheckResult`]

Checks whether records in this table can be expanded.

Returns `{hasPermission: true}` if records can be expanded, `{hasPermission: false, reasonDisplayString: st`
otherwise.

```
const expandRecordsCheckResult = table.checkPermissionToExpan
if (!expandRecordsCheckResult.hasPermission) {
    alert(expandRecordsCheckResult.reasonDisplayString);
}
```

`checkPermissionsForCreateRecord`

`function (fields?: ObjectMap<`[`FieldId`]` | string`
`void>) =>` [`PermissionCheckResult`]

`fields` object mapping `FieldId` or field name to value for that

Checks whether the current user has permission to create the specifi

Accepts partial input, in the same format as [createRecordAsync]. Th
information provided, the more accurate the permissions check will

Returns `{hasPermission: true}` if the current user can creat
record, `{hasPermission: false, reasonDisplayStri`
otherwise. `reasonDisplayString` may be used to display an e

the user.

```
// Check if user can create a specific record, when you alre
// fields/cell values will be set for the record.
const createRecordCheckResult = table.checkPermissionsForCre
    'Project Name': 'Advertising campaign',
    'Budget': 100,
});
if (!createRecordCheckResult.hasPermission) {
    alert(createRecordCheckResult.reasonDisplayString);
}

// Like createRecordAsync, you can use either field names or
const checkResultWithFieldIds = table.checkPermissionsForCre
    [projectNameField.id]: 'Cat video',
    [budgetField.id]: 200,
});

// Check if user could potentially create a record.
// Use when you don't know the specific fields/cell values y
// to show or hide UI controls that let you start creating a
const createUnknownRecordCheckResult =
    table.checkPermissionsForCreateRecord();
```

checkPermissionsForCreateRecords

```
function (records?: ReadonlyArray<{
fields?: ObjectMap<FieldId | string, unknown
void;
}>) => PermissionCheckResult
```

`records`	Array of objects mapping `FieldId` or field name to

Checks whether the current user has permission to create the speci

Accepts partial input, in the same format as createRecordsAsync.
information provided, the more accurate the permissions check wi

Returns `{hasPermission: true}` if the current user can cre
records, `{hasPermission: false, reasonDisplaySt`
otherwise. `reasonDisplayString` may be used to display an
the user.

```
// Check if user can create specific records, when you alre
// fields/cell values will be set for the records.
const createRecordsCheckResult = table.checkPermissionsForC
    // Like createRecordsAsync, fields can be specified by
    {
        fields: {
            'Project Name': 'Advertising campaign',
            'Budget': 100,
        },
    },
    {
        fields: {
            [projectNameField.id]: 'Cat video',
            [budgetField.id]: 200,
        },
    },
    {},
]);
if (!createRecordsCheckResult.hasPermission) {
    alert(createRecordsCheckResult.reasonDisplayString);
}

// Check if user could potentially create records.
// Use when you don't know the specific fields/cell values
// to show or hide UI controls that let you start creating
// Equivalent to table.checkPermissionsForCreateRecord()
const createUnknownRecordCheckResult =
    table.checkPermissionsForCreateRecords();
```

checkPermissionsForDeleteRecord

`function (recordOrRecordId?: Record | RecordI`
`PermissionCheckResult`

`recordOrRecordId`	the record to be deleted

Checks whether the current user has permission to delete the specifi

Accepts optional input, in the same format as deleteRecordAsync. T
information provided, the more accurate the permissions check will

Returns `{hasPermission: true}` if the current user can delet
record, `{hasPermission: false, reasonDisplayStri`
otherwise. `reasonDisplayString` may be used to display an e
user.

```
// Check if user can delete a specific record
const deleteRecordCheckResult =
    table.checkPermissionsForDeleteRecord(record);
if (!deleteRecordCheckResult.hasPermission) {
    alert(deleteRecordCheckResult.reasonDisplayString);
}

// Check if user could potentially delete a record.
// Use when you don't know the specific record you want to d
// example, to show/hide UI controls that let you select a r
const deleteUnknownRecordCheckResult =
    table.checkPermissionsForDeleteRecord();
```

`checkPermissionsForDeleteRecords`

`function (recordsOrRecordIds?: ReadonlyArray`
`RecordId>) => PermissionCheckResult`

`recordsOrRecordIds`	the records to be deleted

Checks whether the current user has permission to delete the speci

Accepts optional input, in the same format as deleteRecordsAsync
information provided, the more accurate the permissions check wi

Returns `{hasPermission: true}` if the current user can del
records, `{hasPermission: false, reasonDisplaySt`
otherwise. `reasonDisplayString` may be used to display an
the user.

```
// Check if user can delete specific records
const deleteRecordsCheckResult =
    table.checkPermissionsForDeleteRecords([record1, recor
if (!deleteRecordsCheckResult.hasPermission) {
    alert(deleteRecordsCheckResult.reasonDisplayString);
}

// Check if user could potentially delete records.
// Use when you don't know the specific records you want to
// example, to show/hide UI controls that let you select re
// Equivalent to table.hasPermissionToDeleteRecord()
const deleteUnknownRecordsCheckResult =
    table.checkPermissionsForDeleteRecords();
```

`checkPermissionsForUpdateRecord`

`function (recordOrRecordId?: Record | RecordI`
`ObjectMap<FieldId | string, unknown | void>)`
`PermissionCheckResult`

`recordOrRecordId`	the record to update
`fields`	cell values to update in that record, specifi `FieldId` or field name to value for that f

Checks whether the current user has permission to perform the give

Accepts partial input, in the same format as updateRecordAsync. Th
provided, the more accurate the permissions check will be.

Returns `{hasPermission: true}` if the current user can upda
`{hasPermission: false, reasonDisplayString: s`
`reasonDisplayString` may be used to display an error messag

```
// Check if user can update specific fields for a specific r
const updateRecordCheckResult =
    table.checkPermissionsForUpdateRecord(record, {
        'Post Title': 'How to make: orange-mango pound cake
        'Publication Date': '2020-01-01',
    });
if (!updateRecordCheckResult.hasPermission) {
    alert(updateRecordCheckResult.reasonDisplayString);
}

// Like updateRecordAsync, you can use either field names or
const updateRecordCheckResultWithFieldIds =
    table.checkPermissionsForUpdateRecord(record, {
        [postTitleField.id]: 'Cake decorating tips & tricks
        [publicationDateField.id]: '2020-02-02',
    });

// Check if user could update a given record, when you don't
// specific fields that will be updated yet (e.g. to check w
// allow a user to select a certain record to update).
const updateUnknownFieldsCheckResult =
    table.checkPermissionsForUpdateRecord(record);

// Check if user could update specific fields, when you don'
// specific record that will be updated yet. (for example, i
// selected by the user and you want to check if your extens
const updateUnknownRecordCheckResult =
    table.checkPermissionsForUpdateRecord(undefined, {
        'My field name': 'updated value',
        // You can use undefined if you know you're going t
        // but don't know the new cell value yet.
        'Another field name': undefined,
    });

// Check if user could perform updates within the table, wit
// specific record or fields that will be updated yet (e.g.,
// extension in "read only" mode).
const updateUnknownRecordAndFieldsCheckResult =
    table.checkPermissionsForUpdateRecord();
```

`checkPermissionsForUpdateRecords`	`function (records?: ReadonlyArray<{ fields?: ObjectMap<FieldId | string, unknown void; id?: RecordId | void; }>) => PermissionCheckResult` `records` Array of objects containing recordId and fields/cellVal that record (specified as an object mapping `FieldId` cell value) Checks whether the current user has permission to perform the giv

updates.

Accepts partial input, in the same format as updateRecordsAsync.
information provided, the more accurate the permissions check wi

Returns `{hasPermission: true}` if the current user can upd
records, `{hasPermission: false, reasonDisplayStr`
`string}` otherwise. `reasonDisplayString` may be used to
message to the user.

```
const recordsToUpdate = [
    {
        // Validating a complete record update
        id: record1.id,
        fields: {
            'Post Title': 'How to make: orange-mango pounc
            'Publication Date': '2020-01-01',
        },
    },
    {
        // Like updateRecordsAsync, fields can be specifie
        id: record2.id,
        fields: {
            [postTitleField.id]: 'Cake decorating tips & t
            [publicationDateField.id]: '2020-02-02',
        },
    },
    {
        // Validating an update to a specific record, not
        // fields will be updated
        id: record3.id,
    },
    {
        // Validating an update to specific cell values, n
        // record will be updated
        fields: {
            'My field name': 'updated value for unknown re
            // You can use undefined if you know you're go
            // field, but don't know the new cell value ye
            'Another field name': undefined,
        },
    },
];

const updateRecordsCheckResult =
    table.checkPermissionsForUpdateRecords(recordsToUpdate)
if (!updateRecordsCheckResult.hasPermission) {
    alert(updateRecordsCheckResult.reasonDisplayString);
}
```

```
// Check if user could potentially update records.
// Equivalent to table.checkPermissionsForUpdateRecord()
const updateUnknownRecordAndFieldsCheckResult =
    table.checkPermissionsForUpdateRecords();
```

createRecordAsync

`function (fields: ObjectMap<FieldId | string, unknown> = {}) => Promise<RecordId>`

`fields`	object mapping `FieldId` or field name to value for that field.

Creates a new record with the specified cell values.

Throws an error if the user does not have permission to create the given records, or if invalid input is provided (eg. invalid cell values).

Refer to FieldType for cell value write formats.

This action is asynchronous: `await` the returned promise if you wish to wait for the new record to be persisted to Airtable servers. Updates are applied optimistically locally, so your changes will be reflected in your extension before the promise resolves.

The returned promise will resolve to the RecordId of the new record once it is persisted.

```
function createNewRecord(recordFields) {
    if (table.hasPermissionToCreateRecord(recordFields)) {
        table.createRecordAsync(recordFields);
    }
    // You can now access the new record in your extension (eg
    // `table.selectRecords()`) but it is still being saved to Airtable
    // servers (e.g. other users may not be able to see it yet).
}

async function createNewRecordAsync(recordFields) {
    if (table.hasPermissionToCreateRecord(recordFields)) {
        const newRecordId = await table.createRecordAsync(recordFields);
    }
    // New record has been saved to Airtable servers.
    alert(`new record with ID ${newRecordId} has been created`);
}

// Fields can be specified by name or ID
createNewRecord({
    'Project Name': 'Advertising campaign',
    'Budget': 100,
});
createNewRecord({
    [projectNameField.id]: 'Cat video',
    [budgetField.id]: 200,
});

// Cell values should generally have format matching the output of
// record.getCellValue() for the field being updated
createNewRecord({
    'Project Name': 'Cat video 2'
    'Category (single select)': {name: 'Video'},
    'Tags (multiple select)': [{name: 'Cats'}, {id: 'someChoiceId'}],
    'Assets (attachment)': [{url: 'http://mywebsite.com/cats.mp4'}],
    'Related projects (linked records)': [{id: 'someRecordId'}],
});
```

createRecordsAsync

```
function (records: ReadonlyArray<{
fields: ObjectMap<FieldId | string, unknown>;
}>) => Promise<Array<RecordId>>
```

`records`	Array of objects with a `fields` key mapping `FieldId` or field name to value for that field.

Creates new records with the specified cell values.

Throws an error if the user does not have permission to create the given records, or if invalid input is provided (eg. invalid cell values).

Refer to FieldType for cell value write formats.

You may only create up to 50 records in one call to `createRecordsAsync`. See Write back to Airtable for more information about write limits.

This action is asynchronous: `await` the returned promise if you wish to wait for the new record to be persisted to Airtable servers. Updates are applied optimistically locall so your changes will be reflected in your extension before the promise resolves.

The returned promise will resolve to an array of RecordIds of the new records once the new records are persisted.

```
const recordDefs = [
    // Fields can be specified by name or ID
    {
        fields: {
            'Project Name': 'Advertising campaign',
            'Budget': 100,
        },
    },
    {
        fields: {
            [projectNameField.id]: 'Cat video',
                [budgetField.id]: 200,
            },
    },
    // Specifying no fields will create a new record with no cell values set
    {
        fields: {},
    },
    // Cell values should generally have format matching the output of
    // record.getCellValue() for the field being updated
    {
        fields: {
            'Project Name': 'Cat video 2'
            'Category (single select)': {name: 'Video'},
            'Tags (multiple select)': [{name: 'Cats'}, {id: 'choiceId'}],
            'Assets (attachment)': [{url: 'http://mywebsite.com/cats.mp4'}]
            'Related projects (linked records)': [{id: 'someRecordId'}],
        },
    },
];

function createNewRecords() {
    if (table.hasPermissionToCreateRecords(recordDefs)) {
        table.createRecordsAsync(recordDefs);
    }
    // You can now access the new records in your extension (e.g.
    // `table.selectRecords()`) but they are still being saved to Airtable
    // servers (e.g. other users may not be able to see them yet.)
}

async function createNewRecordsAsync() {
    if (table.hasPermissionToCreateRecords(recordDefs)) {
        const newRecordIds = await table.createRecordsAsync(recordDefs);
    }
    // New records have been saved to Airtable servers.
    alert(`new records with IDs ${newRecordIds} have been created`);
}
```

`function (recordOrRecordId: Record | RecordId) => Promise<void>`

`recordOrRecordId` the record to be deleted

`deleteRecordAsync`

Delete the given record.

Throws an error if the user does not have permission to delete the given record.

This action is asynchronous: `await` the returned promise if you wish to wait for the delete to be persisted to Airtable servers. Updates are applied optimistically locally, so your changes will be reflected in your extension before the promise resolves.

```
function deleteRecord(record) {
    if (table.hasPermissionToDeleteRecord(record)) {
        table.deleteRecordAsync(record);
    }
    // The record is now deleted within your extension (eg will not be returne
    // in `table.selectRecords`) but it is still being saved to Airtable
    // servers (e.g. it may not look deleted to other users yet).
}

async function deleteRecordAsync(record) {
    if (table.hasPermissionToDeleteRecord(record)) {
        await table.deleteRecordAsync(record);
    }
    // Record deletion has been saved to Airtable servers.
    alert('record has been deleted');
}
```

deleteRecordsAsync

`function (recordsOrRecordIds: ReadonlyArray<Record | RecordId>) => Promise<void>`

`recordsOrRecordIds` Array of Records and RecordIds

Delete the given records.

Throws an error if the user does not have permission to delete the given records.

You may only delete up to 50 records in one call to `deleteRecordsAsync`. See Write back to Airtable for more information about write limits.

This action is asynchronous: `await` the returned promise if you wish to wait for the delete to be persisted to Airtable servers. Updates are applied optimistically locally, so your changes will be reflected in your extension before the promise resolves.

```
function deleteRecords(records) {
    if (table.hasPermissionToDeleteRecords(records)) {
        table.deleteRecordsAsync(records);
    }
    // The records are now deleted within your extension (eg will not be
    // returned in `table.selectRecords()`) but are still being saved to
    // Airtable servers (e.g. they may not look deleted to other users yet).
}

async function deleteRecordsAsync(records) {
    if (table.hasPermissionToDeleteRecords(records)) {
        await table.deleteRecordsAsync(records);
    }
    // Record deletions have been saved to Airtable servers.
    alert('records have been deleted');
}
```

`getField`

`function (fieldIdOrName: FieldId | string) => Field`

`fieldIdOrName` The ID or name of the field you're looking for.

The field matching the given ID or name. Throws if no matching field exists within this table. Use getFieldIfExists instead if you are unsure whether a field exists with the given name/ID.

This method is convenient when building an extension for a specific base, but for more generic extensions the best practice is to use the getFieldById or getFieldByName methods instead.

`getFieldById`

`function (fieldId: FieldId) => Field`

`fieldId` The ID of the field.

Gets the field matching the given ID. Throws if that field does not exist in this table. Use getFieldByIdIfExists instead if you are unsure whether a field exists with the given ID.

```
const fieldId = 'fldxxxxxxxxxxxxxx';
const field = myTable.getFieldById(fieldId);
console.log(field.name);
// => 'Name'
```

`getFieldByIdIfExists`

`function (fieldId: FieldId) => Field | null`

`fieldId` The ID of the field.

Gets the field matching the given ID, or `null` if that field does not exist in this table.

```
const fieldId = 'fldxxxxxxxxxxxxxx';
const field = myTable.getFieldByIdIfExists(fieldId);
if (field !== null) {
    console.log(field.name);
} else {
    console.log('No field exists with that ID');
}
```

`getFieldByName`

`function (fieldName: string) => Field`

`fieldName` The name of the field you're looking for.

Gets the field matching the given name. Throws if no field exists with that name in this table. Use getFieldByNameIfExists instead if you are unsure whether a field exists with the given name.

```
const field = myTable.getFieldByName('Name');
console.log(field.id);
// => 'fldxxxxxxxxxxxxxx'
```

getFieldByNameIfExists	`function (fieldName: string) => Field | null`
	`fieldName` The name of the field you're looking for.
	Gets the field matching the given name, or `null` if no field exists with that name in this table.
	`const field = myTable.getFieldByNameIfExists('Name');`
	`if (field !== null) {`
	`    console.log(field.id);`
	`} else {`
	`    console.log('No field exists with that name');`
	`}`

getFieldIfExists	`function (fieldIdOrName: FieldId | string) => Field | null` `fieldIdOrName` The ID or name of the field you're looking for. The field matching the given ID or name. Returns `null` if no matching field exists within this table. This method is convenient when building an extension for a specific base, but for more generic extensions the best practice is to use the getFieldByIdIfExists or getFieldByNameIfExists methods instead.

hasPermissionToCreateRecord	`function (fields?: ObjectMap<FieldId | string, un`
	`void>) => boolean`
	`fields` object mapping `FieldId` or field name to value for that field.
	An alias for `checkPermissionsForCreateRecord(fields).hasPermis`
	Checks whether the current user has permission to create the specified rec
	Accepts partial input, in the same format as createRecordAsync. The more information provided, the more accurate the permissions check will be.
	`// Check if user can create a specific record, when you already kn`
	`// fields/cell values will be set for the record.`
	`const canCreateRecord = table.hasPermissionToCreateRecord({`

```
    'Project Name': 'Advertising campaign',
    'Budget': 100,
});
if (!canCreateRecord) {
    alert('not allowed!');
}

// Like createRecordAsync, you can use either field names or fiel
const canCreateRecordWithFieldIds = table.hasPermissionToCreateRec
    [projectNameField.id]: 'Cat video',
    [budgetField.id]: 200,
});

// Check if user could potentially create a record.
// Use when you don't know the specific fields/cell values yet (fo
// to show or hide UI controls that let you start creating a recor
const canCreateUnknownRecord = table.hasPermissionToCreateRecord()
```

hasPermissionToCreateRecords

`function (records?: ReadonlyArray<{`
`fields?: ObjectMap<FieldId | string, unknown | v`
`}>) => boolean`

`records`	Array of objects mapping `FieldId` or field name to value f

An alias for `checkPermissionsForCreateRecords(records`

Checks whether the current user has permission to create the specified re

Accepts partial input, in the same format as createRecordsAsync. The m
more accurate the permissions check will be.

```
// Check if user can create specific records, when you already k
// will be set for the records.
const canCreateRecords = table.hasPermissionToCreateRecords([
    // Like createRecordsAsync, fields can be specified by name
    {
        fields: {
            'Project Name': 'Advertising campaign',
            'Budget': 100,
        }
    },
    {
        fields: {
            [projectNameField.id]: 'Cat video',
            [budgetField.id]: 200,
        }
    },
    {},
]);
```

```
if (!canCreateRecords) {
    alert('not allowed');
}

// Check if user could potentially create records.
// Use when you don't know the specific fields/cell values yet (
// to show or hide UI controls that let you start creating recor
// Equivalent to table.hasPermissionToCreateRecord()
const canCreateUnknownRecords = table.hasPermissionToCreateRecor
```

`hasPermissionToDeleteRecord`	`function (recordOrRecordId?: Record | RecordId) =` `recordOrRecordId` the record to be deleted An alias for `checkPermissionsForDeleteRecord(recordOrRecordId)` `ssion`. Checks whether the current user has permission to delete the specified rec Accepts optional input, in the same format as deleteRecordAsync. The mo information provided, the more accurate the permissions check will be.

```
// Check if user can delete a specific record
const canDeleteRecord = table.hasPermissionToDeleteRecord(record);
if (!canDeleteRecord) {
    alert('not allowed');
}

// Check if user could potentially delete a record.
// Use when you don't know the specific record you want to delete
// example, to show/hide UI controls that let you select a record
const canDeleteUnknownRecord = table.hasPermissionToDeleteRecord()
```

`hasPermissionToDeleteRecords`	`function (recordsOrRecordIds?: ReadonlyArray<Rec` `RecordId>) => boolean` `recordsOrRecordIds` the records to be deleted An alias for `checkPermissionsForDeleteRecords(recordsOrRecord` `ermission`. Checks whether the current user has permission to delete the specified re Accepts optional input, in the same format as deleteRecordsAsync. The information provided, the more accurate the permissions check will be.

```
// Check if user can delete specific records
```

```
const canDeleteRecords =
    table.hasPermissionToDeleteRecords([record1, record2]);
if (!canDeleteRecords) {
    alert('not allowed!');
}

// Check if user could potentially delete records.
// Use when you don't know the specific records you want to dele
// example, to show/hide UI controls that let you select records
// Equivalent to table.hasPermissionToDeleteRecord()
const canDeleteUnknownRecords = table.hasPermissionToDeleteRecor
```

`function () => boolean`

An alias for `checkPermissionsForExpandRecords().hasPermission`

`hasPermissionToExpandRecords`

Whether records in this table can be expanded.

```
const isRecordExpansionEnabled = table.hasPermissionToExpandReco
if (isRecordExpansionEnabled) {
    expandRecord(record);
}
```

`function (recordOrRecordId?: Record | RecordId, f`
`ObjectMap<FieldId | string, unknown | void>) => b`

`recordOrRecordId`	the record to update
`fields`	cell values to update in that record, specified as mapping `FieldId` or field name to value for th

An alias for `checkPermissionsForUpdateRecord(recordOrR` `fields).hasPermission`.

`hasPermissionToUpdateRecord`

Checks whether the current user has permission to perform the given reco

Accepts partial input, in the same format as updateRecordAsync. The mor information provided, the more accurate the permissions check will be.

```
// Check if user can update specific fields for a specific record.
const canUpdateRecord = table.hasPermissionToUpdateRecord(record,
    'Post Title': 'How to make: orange-mango pound cake',
    'Publication Date': '2020-01-01',
});
if (!canUpdateRecord) {
    alert('not allowed!');
}
```

```
// Like updateRecordAsync, you can use either field names or fiel
const canUpdateRecordWithFieldIds =
    table.hasPermissionToUpdateRecord(record, {
        [postTitleField.id]: 'Cake decorating tips & tricks',
        [publicationDateField.id]: '2020-02-02',
    });

// Check if user could update a given record, when you don't know
// specific fields that will be updated yet (e.g. to check whether
// allow a user to select a certain record to update).
const canUpdateUnknownFields = table.hasPermissionToUpdateRecord(r

// Check if user could update specific fields, when you don't know
// specific record that will be updated yet (e.g. if the field is
// by the user and you want to check if your extension can write t
const canUpdateUnknownRecord =
    table.hasPermissionToUpdateRecord(undefined, {
        'My field name': 'updated value',
        // You can use undefined if you know you're going to upda
        // but don't know the new cell value yet.
        'Another field name': undefined,
    });

// Check if user could perform updates within the table, without k
// specific record or fields that will be updated yet. (for exampl
// render your extension in "read only" mode)
const canUpdateUnknownRecordAndFields = table.hasPermissionToUpdat
```

```
function (records?: ReadonlyArray<{
fields?: ObjectMap<FieldId | string, unknown | v
void;
id?: RecordId | void;
}>) => boolean
```

`records`	Array of objects containing recordId and fields/cellValues to that record (specified as an object mapping `FieldId` or fie value)

`hasPermissionToUpdateRecords`

An alias for `checkPermissionsForUpdateRecords(records).hasPer`

Checks whether the current user has permission to perform the given rec

Accepts partial input, in the same format as updateRecordsAsync. The m information provided, the more accurate the permissions check will be.

```
const recordsToUpdate = [
    {
        // Validating a complete record update
        id: record1.id,
        fields: {
            'Post Title': 'How to make: orange-mango pound cake
            'Publication Date': '2020-01-01',
        },
    },
    {
        // Like updateRecordsAsync, fields can be specified by 
        id: record2.id,
        fields: {
            [postTitleField.id]: 'Cake decorating tips & tricks
            [publicationDateField.id]: '2020-02-02',
        },
    },
    {
        // Validating an update to a specific record, not knowi
        // fields will be updated
        id: record3.id,
    },
    {
        // Validating an update to specific cell values, not kn
        // record will be updated
        fields: {
            'My field name': 'updated value for unknown record'
            // You can use undefined if you know you're going t
            // field, but don't know the new cell value yet.
            'Another field name': undefined,
        },
    },
];

const canUpdateRecords = table.hasPermissionToUpdateRecords(reco
if (!canUpdateRecords) {
    alert('not allowed');
}

// Check if user could potentially update records.
// Equivalent to table.hasPermissionToUpdateRecord()
const canUpdateUnknownRecordsAndFields =
    table.hasPermissionToUpdateRecords();
```

`toString` `function () => string`

A string representation of the model for use in debugging.

`unwatch` `function (keys: WatchableTableKeyCore | WatchableKeys | ReadonlyArray<WatchableTableKeyCore | WatchableKeys>,` `callback: function (model: this, key: WatchableTableKeyCore | WatchableKeys, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableTableKeyCore | WatchableKeys>`

`keys`	the keys to unwatch
`callback`	the function passed to `.watch` for these keys
`context`	the context that was passed to `.watch` for this `callback`

Unwatch keys watched with `.watch`.

Should be called with the same arguments given to `.watch`.

Returns the array of keys that were unwatched.

`updateRecordAsync`

`function (recordOrRecordId: Record | RecordId, fields: ObjectMap<FieldId | string, unknown>) => Promise<void>`

`recordOrRecordId`	the record to update
`fields`	cell values to update in that record, specified as object mapping `FieldId` or field name to value for that field.

Updates cell values for a record.

Throws an error if the user does not have permission to update the given cell values in the record, or if invalid input is provided (eg. invalid cell values).

Refer to FieldType for cell value write formats.

This action is asynchronous: `await` the returned promise if you wish to wait for the updated cell values to be persisted to Airtable servers. Updates are applied optimistically locally, so your changes will be reflected in your extension before the promise resolves.

```
function updateRecord(record, recordFields) {
    if (table.hasPermissionToUpdateRecord(record, recordFields)) {
        table.updateRecordAsync(record, recordFields);
    }
    // The updated values will now show in your extension (eg in
    // `table.selectRecords()` result) but are still being saved to Airtable
    // servers (e.g. other users may not be able to see them yet).
}

async function updateRecordAsync(record, recordFields) {
    if (table.hasPermissionToUpdateRecord(record, recordFields)) {
        await table.updateRecordAsync(record, recordFields);
    }
    // New record has been saved to Airtable servers.
    alert(`record with ID ${record.id} has been updated`);
}

// Fields can be specified by name or ID
updateRecord(record1, {
    'Post Title': 'How to make: orange-mango pound cake',
    'Publication Date': '2020-01-01',
});
updateRecord(record2, {
    [postTitleField.id]: 'Cake decorating tips & tricks',
    [publicationDateField.id]: '2020-02-02',
});

// Cell values should generally have format matching the output of
// record.getCellValue() for the field being updated
updateRecord(record1, {
    'Category (single select)': {name: 'Recipe'},
    'Tags (multiple select)': [{name: 'Desserts'}, {id: 'someChoiceId'}],
    'Images (attachment)': [{url: 'http://mywebsite.com/cake.png'}],
    'Related posts (linked records)': [{id: 'someRecordId'}],
});
```

`updateRecordsAsync`

```
function (records: ReadonlyArray<{
fields: ObjectMap<FieldId | string, unknown>;
id: RecordId;
}>) => Promise<void>
```

`records`	Array of objects containing recordId and fields/cellValues to update for tha record (specified as an object mapping `FieldId` or field name to cell valu

Updates cell values for records.

Throws an error if the user does not have permission to update the given cell values in t records, or if invalid input is provided (eg. invalid cell values).

Refer to FieldType for cell value write formats.

You may only update up to 50 records in one call to `updateRecordsAsync`. See W back to Airtable for more information about write limits.

This action is asynchronous: `await` the returned promise if you wish to wait for the updates to be persisted to Airtable servers. Updates are applied optimistically locally, so your changes will be reflected in your extension before the promise resolves.

```
const recordsToUpdate = [
    // Fields can be specified by name or ID
    {
        id: record1.id,
        fields: {
            'Post Title': 'How to make: orange-mango pound cake',
            'Publication Date': '2020-01-01',
        },
    },
    {
        id: record2.id,
        fields: {
            // Sets the cell values to be empty.
            'Post Title': '',
            'Publication Date': '',
        },
    },
    {
        id: record3.id,
        fields: {
            [postTitleField.id]: 'Cake decorating tips & tricks',
            [publicationDateField.id]: '2020-02-02',
        },
    },
    // Cell values should generally have format matching the output of
    // record.getCellValue() for the field being updated
    {
        id: record4.id,
        fields: {
            'Category (single select)': {name: 'Recipe'},
            'Tags (multiple select)': [{name: 'Desserts'}, {id: 'choiceId'}]
            'Images (attachment)': [{url: 'http://mywebsite.com/cake.png'}],
            'Related posts (linked records)': [{id: 'someRecordId'}],
        },
    },
];

function updateRecords() {
    if (table.hasPermissionToUpdateRecords(recordsToUpdate)) {
        table.updateRecordsAsync(recordsToUpdate);
    }
    // The records are now updated within your extension (eg will be reflecte
    // `table.selectRecords()`) but are still being saved to Airtable servers
    // (e.g. they may not be updated for other users yet).
}

async function updateRecordsAsync() {
    if (table.hasPermissionToUpdateRecords(recordsToUpdate)) {
        await table.updateRecordsAsync(recordsToUpdate);
    }
    // Record updates have been saved to Airtable servers.
    alert('records have been updated');
```

```
}
```

`watch`	`function (keys: WatchableTableKeyCore | WatchableKeys | ReadonlyArray<WatchableTableKeyCore | WatchableKeys>, callback: function (model: this, key: WatchableTableKeyCore | WatchableKeys, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableTableKeyCore | WatchableKeys>`	
	`keys`	the keys to watch
	`callback`	a function to call when those keys change
	`context`	an optional context for `this` in `callback`.
	Get notified of changes to the model.	
	Every call to `.watch` should have a matching call to `.unwatch`.	
	Returns the array of keys that were watched.	

AbstractModel

View source

Abstract superclass for all models. You won't use this class directly.

Members

`class AbstractModel extends Watchable<WatchableKey>`

`readonly id`	`string` The ID for this model.
`readonly isDeleted`	`boolean` `true` if the model has been deleted, and `false` otherwise. In general, it's best to avoid keeping a reference to an object past the current event loop, since it may be deleted and trying to access any data of a deleted object (other than its ID) will throw. But if you keep a reference, you can use `isDeleted` to check that it's safe to access the model's data.
`toString`	`function () => string` A string representation of the model for use in debugging.

`unwatch`	`function (keys: WatchableKey | ReadonlyArray<WatchableKey>, callback: function (model: this, key: WatchableKey, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableKey>`
	
	`keys` the keys to unwatch
	`callback` the function passed to `.watch` for these keys
	`context` the context that was passed to `.watch` for this `callback`
	
	Unwatch keys watched with `.watch`.
	
	Should be called with the same arguments given to `.watch`.
	
	Returns the array of keys that were unwatched.

`watch`	`function (keys: WatchableKey | ReadonlyArray<WatchableKey>, callback: function (model: this, key: WatchableKey, args: ...Array<any>) => unknown, context?: FlowAnyObject | null) => Array<WatchableKey>` `keys` — the keys to watch `callback` — a function to call when those keys change `context` — an optional context for `this` in `callback`. Get notified of changes to the model. Every call to `.watch` should have a matching call to `.unwatch`. Returns the array of keys that were watched.

Watchable

View source

Abstract superclass for watchable models. All watchable models expose `watch` and `unwatch` methods that allow consumers to subscribe to changes to that model.

This class should not be used directly.

Members

`class Watchable`

`unwatch`

```
function (keys: WatchableKey | ReadonlyArray<WatchableKey>,
callback: function (model: this, key: WatchableKey,
args: ...Array<any>) => unknown, context?: FlowAnyObject |
null) => Array<WatchableKey>
```

`keys`	the keys to unwatch
`callback`	the function passed to `.watch` for these keys
`context`	the context that was passed to `.watch` for this `callback`

Unwatch keys watched with `.watch`.

Should be called with the same arguments given to `.watch`.

Returns the array of keys that were unwatched.

`watch`

```
function (keys: WatchableKey | ReadonlyArray<WatchableKey>,
callback: function (model: this, key: WatchableKey,
args: ...Array<any>) => unknown, context?: FlowAnyObject |
null) => Array<WatchableKey>
```

`keys`	the keys to watch
`callback`	a function to call when those keys change
`context`	an optional context for `this` in `callback`.

Get notified of changes to the model.

Every call to `.watch` should have a matching call to `.unwatch`.

Returns the array of keys that were watched.

Interface

CreateMultipleRecordsMutation

View source

The Mutation emitted when the App creates one or more Records.

```
interface CreateMultipleRecordsMutation
```

`records`	`ReadonlyArray<{ cellValuesByFieldId: ObjectMap<FieldId, unknown>; id: RecordId; }>` The records being created
`tableId`	`TableId` The identifier for the Table in which Records are being created
`type`	`"createMultipleRecords"` This Mutation's discriminant property

Interface

DeleteMultipleRecordsMutation

View source

The Mutation emitted when the App deletes one or more Records.

```
interface DeleteMultipleRecordsMutation
```

`recordIds`	`ReadonlyArray<RecordId>` The identifiers for records being deleted
`tableId`	`TableId` The identifier for the Table in which Records are being deleted
`type`	`"deleteMultipleRecords"` This Mutation's discriminant property

Interface

SetMultipleGlobalConfigPathsMutation

View source

The Mutation emitted when the App modifies one or more values in the GlobalConfig.

```
interface SetMultipleGlobalConfigPathsMutation
```

`type`	`"setMultipleGlobalConfigPaths"` This Mutation's discriminant property
`updates`	`ReadonlyArray<GlobalConfigUpdate>` One or more pairs of path and value

Interface

SetMultipleRecordsCellValuesMutation

View source

The Mutation emitted when the App modifies one or more Records.

```
interface SetMultipleRecordsCellValuesMutation
```

`records`	`ReadonlyArray<{ cellValuesByFieldId: ObjectMap<FieldId, unknown>; id: RecordId; }>` The Records being modified
`tableId`	`TableId` The identifier for the @link Table in which Records are being modified
`type`	`"setMultipleRecordsCellValues"` This Mutation's discriminant property

React component

CellRenderer

View source

Displays the contents of a cell given a field and record.

Props

`interface CellRendererProps`

`cellClassName`	`undefined | string` Additional class names to apply to the cell itself, separated by spaces.
`cellStyle`	`React.CSSProperties` Additional styles to apply to the cell itself.
`cellValue`	`unknown` The cell value to render. Either `record` or `cellValue` must be provided to the CellRenderer. If both are provided, `record` will be used.
`className`	`undefined | string` Additional class names to apply to the cell renderer container, separated by spaces.
`field`	`Field` The Field for a given Record being rendered as a cell.
`record`	`Record | null | undefined` The Record from which to render a cell. Either `record` or `cellValue` must be provided to the CellRenderer. If both are provided, `record` will be used.
`renderInvalidCellValue`	`undefined | function (cellValue: unknown, field: Field) => ReactElement`

	Render function if provided and validation fails.
`shouldWrap`	`undefined | false | true` Whether to wrap cell contents. Defaults to true.
`style`	`React.CSSProperties` Additional styles to apply to the cell renderer container.

React hook

useBase

View source

A hook for connecting a React component to your base's schema. This returns a Base instance and will re-render your component whenever the base's schema changes. That means any change to your base like tables being added or removed, fields getting renamed, etc. It excludes any change to the actual records in the base.

`useBase` should meet most of your needs for working with base schema. If you need more granular control of when your component updates or want to do anything other than re-render, the lower level useWatchable hook might help.

Returns the current base.

```
import {useBase} from '@airtable/blocks/interface/ui';

// renders a list of tables and automatically updates
function TableList() {
     const base = useBase();

     const tables = base.tables.map(table => {
         return <li key={table.id}>{table.name}</li>;
     });

     return <ul>{tables}</ul>;
}
```

Function signature

`function () => Base`

React hook

useColorScheme

View source

A hook for checking whether Airtable is in light mode or dark mode.

```
import {useColorScheme} from '@airtable/blocks/interface/ui';

function MyApp() {
    const {colorScheme} = useColorScheme();
    return (
        <div style={colorScheme === 'dark' ?
            {color: 'white', backgroundColor: 'black'} :
            {color: 'black', backgroundColor: 'white'}
        }>
            Tada!
        </div>
    );
}
```

Function signature

```
function () => {
colorScheme: "light" | "dark";
}
```

React hook

useCustomProperties

View source

A hook for integrating configuration settings for your block with the Interface Designer properties panel. Under the hood, this uses GlobalConfig to store the custom property values.

Returns an object with:

- `customPropertyValueByKey`: an object mapping custom property keys to their current value.
- `errorState`: an object with an `error` property if there was an error setting the custom properties

```
import {useCustomProperties} from '@airtable/blocks/interface/ui';

function getCustomProperties(base: Base) {
    const table = base.tables[0];
    const isNumberField = (field: {id: FieldId, config: FieldConfig}) => field.config.type
=== FieldType.NUMBER;
    return [
        {key: 'title', label: 'Title', type: 'string', defaultValue: 'Chart'},
        {key: 'xAxis', label: 'X-axis', type: 'field', table, shouldFieldBeAllowed:
isNumberField},
        {key: 'yAxis', label: 'Y-axis', type: 'field', table, shouldFieldBeAllowed:
isNumberField},
        {key: 'color', label: 'Color', type: 'enum', possibleValues: [{value: 'red', label:
'Red'}, {value: 'blue', label: 'Blue'}, {value: 'green', label: 'Green'}], defaultValue:
'red'},
        {key: 'showLegend', label: 'Show Legend', type: 'boolean', defaultValue: true},
    ];
}

function MyApp() {
    const {customPropertyValueByKey, errorState} =
useCustomProperties(getCustomProperties);
}
```

Function signature

```
function (getCustomProperties: function (base: Base) =>
Array<BlockPageElementCustomProperty>) => {
customPropertyValueByKey: {[key: string]: unknown};
errorState: {
error: Error;
} | null;
}
```

`getCustomProperties`	A function that returns an array of BlockPageElementCustomProperty. This function should have a stable identity, so it should either be defined at the top level of the file or wrapped in useCallback. It will receive an instance of Base as an argument.

React hook

useGlobalConfig

View source

Returns the extension's GlobalConfig and updates whenever any key in GlobalConfig changes.

```
import {useGlobalConfig, useRunInfo} from '@airtable/blocks/interface/ui';

function SyncedCounter() {
    const runInfo = useRunInfo();
    const globalConfig = useGlobalConfig();
    const count = globalConfig.get('count');

    const increment = () => globalConfig.setAsync('count', count + 1);
    const decrement = () => globalConfig.setAsync('count', count - 1);
    const isEnabled = globalConfig.hasPermissionToSet('count');

    if (runInfo.isPageElementInEditMode) {
        return (
            <div>
                <button onClick={decrement} disabled={!isEnabled} ariaLabel="decrease">-
</button>
                {count}
                <button onClick={increment} disabled={!isEnabled}
ariaLabel="increase">+</button>
            </div>
        );
    } else {
        return <div>{count}</div>;
    }
}
```

Function signature

`function () => GlobalConfig`

React hook

useRecords

View source

A hook for working with all of the records (including cell values) in a particular table. Automatically handles loading data and updating your component when the underlying data changes.

This hook re-renders when data concerning the records changes (specifically, when cell values change and when records are added or removed).

Returns a list of records.

```
import {useBase, useRecords} from '@airtable/blocks/interface/ui';

function RecordList() {
    const base = useBase();
    const table = base.tables[0];

    // grab all the records from that table
    const records = useRecords(table);

    // render a list of records:
    return (
        <ul>
            {records.map(record => {
                return <li key={record.id}>{record.name}</li>;
            })}
        </ul>
    );
}
```

Function signature

`function (table: Table) => Array<Record>`

`table`	The Table you want the records from.

React hook

useRunInfo

View source

A hook for getting information about the current run context. This can be useful if you'd like to display some configuration options when the page element is in edit mode.

`useRunInfo`

```
import {useRunInfo} from '@airtable/blocks/interface/ui';

// renders a list of tables and automatically updates
function MyApp() {
     const runInfo = useRunInfo();
     return (
         <div>
             <p>Is development mode: {runInfo.isDevelopmentMode ? 'Yes' : 'No'}</p>
             <p>Is page element in edit mode: {runInfo.isPageElementInEditMode ? 'Yes' :
'No'}</p>
         </div>
     );
}
```

Function signature

```
function () => {
isDevelopmentMode: boolean;
isPageElementInEditMode: boolean;
}
```

React hook

useSession

View source

A hook for connecting a React component to the current session. This returns a Session instance and will re-render your component whenever the session changes (e.g. when the current user's permissions change or when the current user's name changes).

`useSession` should meet most of your needs for working with Session. If you need more granular control of when your component updates or want to do anything other than re-render, the lower level useWatchable hook might help.

```
import {useSession} from '@airtable/blocks/interface/ui';

// Says hello to the current user and updates in realtime if the current user's
// name or profile pic changes.
function CurrentUserGreeter() {
    const session = useSession();
    return (
        <React.Fragment>
            Hello {session.currentUser?.name ?? 'stranger'}!
        </React.Fragment>
    );
}
```

Function signature

`function () => Session`

React hook

useSynced

View source

A hook for syncing a component to GlobalConfig. Useful if you are dealing with a custom input component and can‘t use one of our `Synced` components.

```
import {useBase, useSynced} from '@airtable/blocks/interface/ui';

function CustomInputSynced() {
   const [value, setValue, canSetValue] = useSynced('myGlobalConfigKey');

    return (
        <input
             type="text"
             value={value}
             onChange={e => setValue(e.target.value)}
             disabled={!canSetValue}
         />
    );
}
```

Function signature

`function (globalConfigKey: GlobalConfigKey) => []`

React hook

useWatchable

View source

A React hook for watching data in Airtable models like Table and Record. Each model has several watchable keys that can be used with this hook to have your component automatically re-render when data in the models changes. You can also provide an optional callback if you need to do anything other than re-render when the data changes.

This is a low-level tool that you should only use when you specifically need it. There are more convenient model-specific hooks available:

- For Base, Table, or Field, use useBase.
- For Record, use useRecords, useRecordIds, or useRecordById.

If you're writing a class component and still want to be able to use hooks, try withHooks.

```
import {useWatchable} from '@airtable/blocks/interface/ui';

function TableName({table}) {
    useWatchable(table, 'name');
    return <span>The table name is {table.name}</span>;
}

function RecordValues({record, field}) {
    useWatchable(record, ['cellValues']);
    return <span>
        The record has cell value {record.getCellValue(field)} in {field.name}.
    </span>
}
```

Function signature

`function (models: Watchable<Keys> | ReadonlyArray<Watchable<Keys> | null | undefined> | null | undefined, keys: Keys | ReadonlyArray<Keys | null> | null, callback?: undefined | function (model: Watchable<Keys>, keys: string, args: ...Array<any>) => unknown) => void`

`models`	The model or models to watch.
`keys`	The key or keys to watch. Non-optional, but may be null.
`callback`	An optional callback to call when any of the watch keys change.

Colors

View source

Airtable color names.

To get the corresponding RGB or HEX values, use getRgbForColor or getHexForColor.

You can also pass these values into the color props for components:

```
import {colors, colorUtils} from '@airtable/blocks/interface/ui';

<div style={{backgroundColor: colorUtils.getHexForColor(colors.BLUE)}}>Hello world</div>
```

Properties

BLUE	"blue"
BLUE_BRIGHT	"blueBright"
BLUE_DARK_1	"blueDark1"
BLUE_LIGHT_1	"blueLight1"
BLUE_LIGHT_2	"blueLight2"
CYAN	"cyan"
CYAN_BRIGHT	"cyanBright"
CYAN_DARK_1	"cyanDark1"
CYAN_LIGHT_1	"cyanLight1"
CYAN_LIGHT_2	"cyanLight2"
GRAY	"gray"
GRAY_BRIGHT	"grayBright"
GRAY_DARK_1	"grayDark1"
GRAY_LIGHT_1	"grayLight1"
GRAY_LIGHT_2	"grayLight2"
GREEN	"green"
GREEN_BRIGHT	"greenBright"
GREEN_DARK_1	"greenDark1"
GREEN_LIGHT_1	"greenLight1"
GREEN_LIGHT_2	"greenLight2"
ORANGE	"orange"
ORANGE_BRIGHT	"orangeBright"
ORANGE_DARK_1	"orangeDark1"
ORANGE_LIGHT_1	"orangeLight1"
ORANGE_LIGHT_2	"orangeLight2"

PINK	"pink"
PINK_BRIGHT	"pinkBright"
PINK_DARK_1	"pinkDark1"
PINK_LIGHT_1	"pinkLight1"
PINK_LIGHT_2	"pinkLight2"
PURPLE	"purple"
PURPLE_BRIGHT	"purpleBright"
PURPLE_DARK_1	"purpleDark1"
PURPLE_LIGHT_1	"purpleLight1"
PURPLE_LIGHT_2	"purpleLight2"
RED	"red"
RED_BRIGHT	"redBright"
RED_DARK_1	"redDark1"
RED_LIGHT_1	"redLight1"
RED_LIGHT_2	"redLight2"
TEAL	"teal"
TEAL_BRIGHT	"tealBright"
TEAL_DARK_1	"tealDark1"
TEAL_LIGHT_1	"tealLight1"
TEAL_LIGHT_2	"tealLight2"
YELLOW	"yellow"
YELLOW_BRIGHT	"yellowBright"
YELLOW_DARK_1	"yellowDark1"
YELLOW_LIGHT_1	"yellowLight1"
YELLOW_LIGHT_2	"yellowLight2"

Interface

ColorUtils

View source

Utilities for working with Color names from the colors enum.

```
interface ColorUtils
```

`getHexForColor`	`function (colorString: Color) => string`
	`function (colorString: string) => null | string`
	
	Given a Color, return the hex color value for that color, or null if the value isn't a Color
	
	`import {colorUtils, colors} from '@airtable/blocks/interface/ui';`
	
	`colorUtils.getHexForColor(colors.RED);`
	`// => '#ef3061'`
	
	`colorUtils.getHexForColor('uncomfortable beige');`
	`// => null`

`getRgbForColor`	`function (colorString: Color) => RGB`
	`function (colorString: string) => RGB | null`
	
	Given a Color, return an RGB object representing it, or null if the value isn't a Color
	
	`import {colorUtils, colors} from '@airtable/blocks/interface/ui';`
	
	`colorUtils.getRgbForColor(colors.PURPLE_DARK_1);`
	`// => {r: 107, g: 28, b: 176}`
	
	`colorUtils.getRgbForColor('disgruntled pink');`
	`// => null`

`shouldUseLightTextOnColor`	`function (colorString: string) => boolean`
	
	Given a Color, returns true or false to indicate whether that color should have light text on top of it when used as a background color.
	
	`import {colorUtils, colors} from '@airtable/blocks/interface/ui';`
	
	`colorUtils.shouldUseLightTextOnColor(colors.PINK_LIGHT_1);`
	`// => false`
	
	`colorUtils.shouldUseLightTextOnColor(colors.PINK_DARK_1);`

```
// => true
```

Function

expandRecord

View source

Expands the given record in the Airtable UI.

```
import {expandRecord} from '@airtable/blocks/interface/ui';

<button onClick={() => expandRecord(record)}>{record.name}</button>
```

Function signature

`function (record: Record) => void`

`record`	The record to expand.

Function

initializeBlock

View source

`initializeBlock` takes the top-level React component in your tree and renders it. It is conceptually similar to `ReactDOM.render`, but takes care of some Extensions-specific things.

```
import {initializeBlock} from '@airtable/blocks/interface/ui';
import React from 'react';

function App() {
    return (
        <div>Hello world 🚀</div>
    );
}

initializeBlock({interface: () => <App />});
```

Function signature

`function (entryPoints: EntryPoints) => void`

`entryPoints`	An object with an `interface` property which is a function that returns your React Node.

Function

loadCSSFromString

View source

Injects CSS from a string into the page. Returns the HTML style element inserted into the page.

```
import {loadCSSFromString} from '@airtable/blocks/interface/ui';
loadCSSFromString('body { background: red; }');
```

Function signature

`function (css: string) => HTMLStyleElement`

`css`	The CSS string.

Function

loadCSSFromURLAsync

View source

Injects CSS from a remote URL.

Returns a promise that resolves to the HTML style element inserted into the page.

```
import {loadCSSFromURLAsync} from '@airtable/blocks/interface/ui';
loadCSSFromURLAsync('https://example.com/style.css');
```

Function signature

`function (url: string) => Promise<HTMLLinkElement>`

`url`	The URL of the stylesheet.

Function

loadScriptFromURLAsync

View source

Injects Javascript from a remote URL.

Returns a promise that resolves to the HTML script element inserted into the page.

```
import {loadScriptFromURLAsync} from '@airtable/blocks/interface/ui';
loadScriptFromURLAsync('https://example.com/script.js');
```

Function signature

`function (url: string) => Promise<HTMLScriptElement>`

`url`	The URL of the script.