# Contributor's Guide

Welcome! We are very excited that you are interested in contributing to Coral.

This document is a companion to help you approach contributing. If it does not

do so, please [let us know how we can improve it](https://github.com/coralproject/talk/issues)!

By contributing to this project you agree to the [Code of Conduct](CODE\_OF\_CONDUCT.md).

<!-- START doctoc generated TOC please keep comment here to allow auto update -->

<!-- DON'T EDIT THIS SECTION, INSTEAD RE-RUN doctoc TO UPDATE -->

## Table of Contents

- [What should I Contribute?](#what-should-i-contribute)

- [Writing Code](#writing-code)

- [When should I create an issue?](#when-should-i-create-an-issue)

- [What should I include in my issue?](#what-should-i-include-in-my-issue)

- [When should I create a pull request?](#when-should-i-create-a-pull-request)

- [What should I include in my pull request?](#what-should-i-include-in-my-pull-request)

- [Reviewing pull requests](#reviewing-pull-requests)

- [Ensure contributions are linted and tested](#ensure-contributions-are-linted-and-tested)

- [Review the feature/fixes](#review-the-featurefixes)

- [Review architectural decisions](#review-architectural-decisions)

- [Verify localizations](#verify-localizations)

- [Localization](#localization)

- [Documentation](#documentation)

- [Design Principles](#design-principles)

- [GraphQL](#graphql)

<!-- END doctoc generated TOC please keep comment here to allow auto update -->

## What should I Contribute?

There are at least three different ways to contribute to Coral:

- [Writing Code](#writing-code)

- [Reviewing pull requests](#reviewing-pull-requests)

- [Localization](#localization)

- [Documentation](#documentation)

Typically these take the form of creating a Pull Request for Coral, and

submitting it to be reviewed by a member of our team and the greater Coral

community.

Working on your first Pull Request? You can learn how from this free video

series:

[How to Contribute to an Open Source Project on GitHub](https://egghead.io/series/how-to-contribute-to-an-open-source-project-on-github)

If you decide to fix an issue, please be sure to check the comment thread in

case somebody is already working on a fix. If nobody is working on it at the

moment, please leave a comment stating that you intend to work on it so other

people don’t accidentally duplicate your effort.

If somebody claims an issue but doesn’t follow up for more than two weeks, it’s

fine to take it over but you should still leave a comment.

## Writing Code

Conversation surrounding contributions begins when you can create an issue

describing your issue or suggestion.

### When should I create an issue?

File an issue as soon as you have an idea of something you'd like to contribute.

We would love to hear what you're thinking and help refine the idea to make it

into the Coral ecosystem.

### What should I include in my issue?

Coral has adopted an iterative, agile development philosophy. All contributions

that make it into the Coral repository should start with a user story in this

form:

> As a [type of Coral user] I'd like to [do something] so that I can [get some result/value].

This exercise does two things:

- Allows us to ground our technical choices in a clear, simple product need.

- Expresses that product need in a way that doesn't imply a specific technical

solution allowing for debate as to the best way to solve the problem.

Please feel free to provide as much detail as possible when filing the issue but

please do keep the initial issue specific to one need and try to avoid including

technical or design solutions.

If you have a specific technical or design solution in mind, please submit it as

the first comment on the thread.

### When should I create a pull request?

File a pull request if you've created an issue in our [issues](https://github.com/coralproject/talk/issues)

page and have heard back from a member or contributor to Coral. This allows our

team to review the proposed changes prior to time being spent if the team

already has the feature/fix in the road map.

### What should I include in my pull request?

When you create a pull request, the template will describe the required

components needed for it to be reviewed by a member of the Coral team. You

should end up filling out:

- What does this PR (pull request) do?

- How do I test this PR?

You should describe what Github issue or ticket that the PR is associated with

to assist the review process. If this PR is resolving a particular bug, a

testing strategy should be described in the testing section. If this PR is

contributing a new feature, a description should describe a scenario to test or

verify the new functionality.

## Reviewing pull requests

Reviewing pull requests in Coral is generally completed by the core Coral team

that is composed of developers employed by Vox Media Inc, but external reviews

or suggestions are also welcomed.

Our review process generally follows a few core principles:

### Ensure contributions are linted and tested

It is the job of CI linting and tests to notify of style issues within the

codebase. If it is not possible for style issues to be encapsulated as a

linting rule, it shouldn't be concretely enforced during the review process.

This can ensure that code reviews contain more meaningful feedback tied to the

contribution rather than nit-picking on stylistic choices.

Reviewers must ensure that linting and tests pass in CI and locally prior to a

review taking place. You can do this by running `npm run generate` followed by

`npm run lint` and `npm run test`.

### Review the feature/fixes

Any new features added to Coral should be reviewed for bugs through a manual

verification process to ensure that they function on your machine. If possible

you should review any automated tests that were added (or not added) related to

the feature.

While the Coral team is not strict on test driven development (or TDD), any

contributions that include tests are greatly appreciated, and preferred over

those that do not.

### Review architectural decisions

Any substantial changes made to the codebase should be reviewed to ensure

that they conform to the current way code/services are laid out.

Architecture Decision Records (or [ADR](http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions)) are now being used to describe architectural decisions and can be found in the `src/docs/architecture/decisions` directory.

### Verify localizations

While we don't have any automated tools at the time of writing that will

verify this in CI, any strings being added that are presented to the end user

should be wrapped in localization components to support other languages.

## Localization

We use the [fluent](http://projectfluent.org/) library and store our

translations in [FTL](http://projectfluent.org/fluent/guide/) files in

`src/locales/` and `src/core/server/locales/`.

Strings are added or removed from localization bundles in the translation files

as needed. Strings \*\*MUST NOT\*\* be \_changed\_ after they've been committed and

pushed to master. Changing a string requires creating a new ID with a new name

(preferably descriptive instead of incremented) and deletion of the obsolete ID.

It's often useful to add a comment above the string with info about how and

where the string is used.

Once a language has enough coverage, it should be added to

`src/core/common/helpers/i18n/locales.ts`.

The [Perspective API](https://github.com/conversationai/perspectiveapi/blob/master/2-api/methods.md#analyzecomment-request)

also supports comments in specific languages. When the language is supported in

Coral and supported by the Perspective API, the language should be added to the

language map in `src/core/server/services/comments/pipeline/phases/toxic.ts`.

To assist with the translation process, we have a script that is based on the

work by @cristiandean in https://github.com/coralproject/talk/pull/2949 that

will detect missing, new, or changed translation keys for the specified

language. You can use this with:

```sh

# usage: ./scripts/i18n/validate.ts <locale>

./scripts/i18n/validate.ts pt-BR

```

## Documentation

Documentation that is publicly shown on

[docs.coralproject.net](https://docs.coralproject.net/coral/) is stored under

the `release/4` branch as it contains information for versions 4.0 onwards of

Coral.

To contribute new docs, you can either click the "Suggest Edits" in the top

right of each page, or you can edit directly via source. We suggest that for

individual fixes or contributions to the documentation.

If you want to contribute via source files, you can follow the procedure

outlined below:

1. Clone the Talk repository via `git clone https://github.com/coralproject/talk.git`

2. Switch to the `release/4` branch via `git checkout release/4`

3. Follow the procedure outlined on that branch's [CONTRIBUTING.md](https://github.com/coralproject/talk/blob/release/4/CONTRIBUTING.md#contributing-documentation) file for contributing documentation changes

4. Create a pull request to merge your changes back into the `release/4` branch

## Design Principles

### GraphQL

Coral relies heavily on [GraphQL](https://graphql.org) as the query language for

the API and the runtime on the server that powers resolving data from data

sources. This heavily influences a lot of the decisions around how we create and

consume it's API internally and how we expose it to others to interact with.

There are many GraphQL types in our [`schema.graphql`](https://github.com/coralproject/talk/blob/master/src/core/server/graph/schema/schema.graphql)

that define the way we handle data in our API. We'll try to outline a few of

them here with examples to help you understand their uses.

#### Types

Similar to defining an interface or a \_struct\_ definition, GraphQL has flexible

types that can be used to define data types that are used for querying data from

the API. This retrieval can happen directly via a query, or after executing an

action using a mutation and querying its response result.

An example of these types is the `Comment` and its nested `CommentRevision`

type:

```graphql

"""

Comment is a comment left by a User on an Story or another Comment as a reply.

"""

type Comment {

"""

id is the identifier of the Comment.

"""

id: ID!

"""

body is the content of the Comment, and is an alias to the body of the

`revision.body`.

"""

body: String

"""

revision is the current revision of the Comment's body.

"""

revision: CommentRevision

"""

revisionHistory stores the previous CommentRevision's, with the most recent

edit last.

"""

revisionHistory: [CommentRevision!]!

@auth(

roles: [MODERATOR, ADMIN]

userIDField: "author\_id"

permit: [SUSPENDED, BANNED, PENDING\_DELETION]

)

"""

status represents the Comment's current status.

"""

status: COMMENT\_STATUS!

}

```

Notice how the `Comment` type can nest more custom defined types. A `Comment`

can have a current `CommentRevision` named `revision`. It also has a list of its

historical `revisionHistory`:

```graphql

type CommentRevision {

"""

id is the identifier of the CommentRevision.

"""

id: ID!

"""

comment is the reference to the original Comment associated with the current

Comment.

"""

comment: Comment!

"""

actionCounts stores the counts of all the actions for the CommentRevision

specifically.

"""

actionCounts: ActionCounts! @auth(roles: [MODERATOR, ADMIN])

"""

body is the content of the CommentRevision. If null, it indicates that the

body text was deleted.

"""

body: String

}

```

Another thing to note, see how `CommentRevision` is not only referenced by

`revision` and `revisionHistory` on the `Comment` type. The `CommentRevision`

also references back to its parent `Comment` via the `comment: Comment!`

property. This is how defined types interact between each other in the GraphQL

schema.

Our naming scheme is upper camel case (also known as Pascal Case) for these

types:

- Start with a capital letter

- Following characters are lower case

- Every new word in the type name begins with a new capital letter

- Acronyms are always capitalized (with the only exception being the

`clientMutationId: String!` field in mutation input/payload types)

Some of the properties have an `!` beside their type (i.e `id: ID!`) which

indicates that this property is required and is non-nullable. GraphQL will

validate the input request for these properties and ensure they are provided

during the GraphQL request.

You can learn more about GraphQL types in their documentation:

[Learn GraphQL: Schemas and Types](https://graphql.org/learn/schema/)

#### Enumeration Types

In the previous example with the `Comment` type. We also had a property called

`status` which was of type `COMMENT\_STATUS`.

This is another kind of defined type, an [Enumeration Type](https://graphql.org/learn/schema/#enumeration-types),

also called \_enums\_.

```graphql

enum COMMENT\_STATUS {

NONE

APPROVED

REJECTED

PREMOD

SYSTEM\_WITHHELD

}

```

Like all enumeration types this definition enumerates out typed, named values

that are reusable for state elsewhere on other types.

Our naming scheme for enumeration types and values in those types as:

- All capital letters

- Spaces delimited with underscores

This is because they are treated as shared constant values across the schema.

Rather than storing strings or numbers to capture selected state, we prefer

using enumeration types because they are much more stricter in terms of value.

You can learn more about GraphQL Enumeration types in their documentation:

[Learn GraphQL: Enumeration Types](https://graphql.org/learn/schema/#enumeration-types)

#### Mutation Types

Mutations are a request to GraphQL to initiate an action which will result in a response. As such,they're broken up into an `Input` and `Payload` pair that matches a mutation's request and response pair.

An example is the `CreateCommentInput` and `CreateCommentPayload`:

```graphql

"""

CreateCommentInput provides the input for the createComment Mutation.

"""

input CreateCommentInput {

"""

storyID is the ID of the Story where we are creating a comment on.

"""

storyID: ID!

"""

nudge when true will instead return an error related to recoverable moderation

faults such as a toxic comment or spam comment to provide user feedback to

nudge the user to correct the comment.

"""

nudge: Boolean = false

"""

body is the Comment body, the content of the Comment.

"""

body: String!

"""

clientMutationId is required for Relay support.

"""

clientMutationId: String!

}

"""

CreateCommentPayload contains the created Comment after the createComment

mutation.

"""

type CreateCommentPayload {

"""

edge is the possibly created comment edge.

"""

edge: CommentEdge!

"""

clientMutationId is required for Relay support.

"""

clientMutationId: String!

}

```

The `CreateCommentInput` type contains some parameters that allow us to execute

the mutation. These are:

- `storyID` - the story we are submitting the comment to.

- `body` - the body of our comment.

- `nudge` - whether we should return validation prompts to the user about

improper comment language.

- `clientMutationId` - the identifier used by Relay to identify this mutation,

our front-end state cache to process the mutation request. This is the only

place in the code-base that we do not capitalize the acronym when written in

camel-case, this is unfortunately due to legacy reasons from within Relay.

The returned response for a mutation is a what we call a Payload, in this case

it's `CreateCommentPayload`. This usually has a response that is the full data

type of whatever was modified by the earlier called mutation action. The

properties on this type are:

- `edge` - we return the full comment edge that was created by the previous

input. This is done so that the mutation request can query whatever it needs

to from the returned input to update state on the client.

- `clientMutationId` - an identifier used by Relay (our front-end client state

cache) to process the mutation request.

#### Access Permissions On Types

Sometimes, you only want specific users to be allowed to view certain

information. For instance, we have roles that are defined in our schema so we

can filter who can have access to what.

These roles are used with directives on our schema that GraphQL then enforces

when trying to resolve requests a user makes. If the user has the specified role

associated with their signed-in account, they are given access to the requested

data.

An example of this is the `revisionHistory` on the `Comment` type:

```graphql

fragment on Comment {

"""

revisionHistory stores the previous CommentRevision's, with the most recent

edit last.

"""

revisionHistory: [CommentRevision!]!

@auth(

roles: [MODERATOR, ADMIN]

userIDField: "author\_id"

permit: [SUSPENDED, BANNED, PENDING\_DELETION]

)

}

```

Here we see the `@auth` directive. It has documentation describing the various

parameters allowed located in the [`schema.graphql`](https://github.com/coralproject/talk/blob/master/src/core/server/graph/schema/schema.graphql)

file, we'll discuss below what this particular set of parameters can be read as:

- The roles that are allowed to access this information are `MODERATOR` and

`ADMIN` as defined by the `roles` argument.

- We let the directive know that the author of the comment is `author\_id` from

the `Comment` by defining the `userIDField`. It's a rule of thumb in Coral if

the Author created the document, they have permission to view it. A Comment

for example is authored by a user, with the underlying field associated with

the id of that author living on the `author\_id` field. You can see how this

is related if you look at the resolver for the `Comment` type to see that it

is based off of the `Comment` interface from `src/core/server/models/comment`.

- We permit returning comments when the author has the following conditions

associated with their account: `SUSPENDED`, `BANNED`, or `PENDING\_DELETION`.

These directives can be simpler, for example the `metadata` property on the

`CommentRevision`:

```graphql

fragment on CommentRevision {

"""

metadata stores details on a CommentRevision.

"""

metadata: CommentRevisionMetadata! @auth(roles: [ADMIN, MODERATOR])

}

```

Here we see an auth directive with only roles defined. This is sufficient to

make sure that the metadata property is only accessible to `ADMIN` and

`MODERATOR` user roles.

Note: Wondering how the user roles are defined? They're simply an enumeration

type that is also defined in the schema.

```graphql

enum USER\_ROLE {

COMMENTER

STAFF

MODERATOR

ADMIN

}

```

#### Arrays of Items

Sometimes you don't want a singular property, your property is instead a

collection of items.

The `revisionHistory` from the `Comment` is again useful as an example:

```graphql

fragment on Comment {

"""

revisionHistory stores the previous CommentRevision's, with the most recent

edit last.

"""

revisionHistory: [CommentRevision!]!

}

```

The interior type `CommentRevision` is required using the `!` and the outer

array is also required using `!`.

We do this for a couple of reasons:

- This ensures that we do not return null/undefined values within the array.

- Why would we return a null when we can just return nothing for null values?

- We want the array to always be defined, if empty, we return and empty array (i.e. `[]`).

- This can be handled nicely in our resolvers. We simply check if the retrieved values is null or undefined and simply return an empty array in its stead.

These little tweaks aren't necessary, but they ease the use of our API by making

the results for arrays predictable and strongly typed.

#### Documenting

As you may have noticed, there is quite a bit of documentation in the schema examples listed here.

We typically follow these two rules in commenting our GraphQL types:

- Always comment the property within a type describing its purpose on its parent type

i.e. `createdAt` on our `Setting` type:

```graphql

fragment on Setting {

"""

createdAt is the time that the Settings was created at.

"""

createdAt: Time! @auth(roles: [ADMIN])

}

```

- Always comment the purpose of each type

i.e. the `Comment` type:

```graphql

"""

Comment is a comment left by a User on an Story or another Comment as a reply.

"""

type Comment { }

```