

---

# **lookit-api Documentation**

***Release 0.0.1***

**Center for Open Science**

**Oct 29, 2019**



<b>1</b>	<b>Using Lookit: for researchers</b>	<b>3</b>
1.1	Getting started guide	3
1.2	Other helpful resources	4
1.3	Using the experimenter interface	5
1.3.1	Logging in	5
1.3.2	Managing Studies	6
1.3.2.1	Viewing study list	6
1.3.2.2	Creating a study	7
1.3.2.3	Study detail page	8
1.3.2.4	Study status	11
1.3.2.5	Adding researchers to your study	11
1.3.2.6	Editing researcher permissions on a study	12
1.3.2.7	Deleting researcher permissions	12
1.3.2.8	Study edit page	13
1.3.2.9	Editing study structure	15
1.3.2.10	Editing study type	16
1.3.2.11	Viewing individual study responses	17
1.3.2.12	Viewing all study responses	18
1.3.2.13	Viewing demographics of study participants	19
1.3.2.14	Viewing all study videos	19
1.3.3	Managing your Organization	19
1.3.3.1	Adding researchers to your organization	20
1.3.3.2	Editing a researcher's organization permissions	20
1.3.3.3	Deleting a researcher's organization permissions	21
1.4	Setting study fields	22
1.4.1	Name	22
1.4.2	Image	22
1.4.3	Short description	22
1.4.4	Purpose	22
1.4.5	Compensation	23
1.4.6	Exit URL	23
1.4.7	Participant eligibility description	23
1.4.8	Criteria expression	23
1.4.8.1	Query fields	24
1.4.8.2	Criteria expression examples	24
1.4.8.3	Characteristics and conditions	24

1.4.8.4	Language codes	24
1.4.9	Minimum and maximum age cutoffs	26
1.4.10	Duration	26
1.4.11	Researcher contact information	26
1.4.12	Discoverable	26
1.4.13	Build study	27
1.4.14	Study type	27
1.5	Building your experiment	27
1.5.1	Preliminaries: JSON format	27
1.5.2	Experiment structure	28
1.5.3	Developing your study: how to try it out as you go	29
1.5.4	Finding and using specific frames	30
1.5.5	A Lookit study schema: general principles and instructions	30
1.5.6	Randomizer frames	31
1.5.6.1	Nested randomizers	36
1.5.7	Conditional logic	40
1.5.7.1	Example: eligibility survey	41
1.5.7.2	Example: waiting for successful training	44
1.5.7.3	Example: personalized story	46
1.5.7.4	Example: debriefing text that depends on experimental condition	46
1.6	Preparing your stimuli	47
1.6.1	Audio and video files	47
1.6.2	File formats	48
1.6.3	Making dummy stimuli	48
1.6.4	Directory structure	49
1.7	Experiment data (non-video)	50
1.7.1	What data can I access?	50
1.7.2	Accessing experiment data	50
1.7.3	Structure of session data	50
1.7.4	Interpreting <code>exp_data</code>	53
1.8	Consent manager	54
1.8.1	Overview	54
1.8.2	Managing consent rulings	55
1.8.2.1	Making consent rulings	57
1.8.2.2	Response statistics	57
1.8.2.3	Withdrawn responses	57
1.9	Using the API	57
1.9.1	What is the API for?	57
1.9.2	API Tips	58
1.9.2.1	General	58
1.9.2.2	API Formatting	58
1.9.2.3	Content-Type	59
1.9.2.4	Authentication	59
1.9.2.5	Pagination	60
1.9.3	Available Endpoints	61
1.9.3.1	Children	61
1.9.3.2	Demographic Data	63
1.9.3.3	Feedback	65
1.9.3.4	Organizations	68
1.9.3.5	Responses	70
1.9.3.6	Studies	73
1.9.3.7	Users	76

## 2 Developing new frames

79

2.1	Setup for custom frame development . . . . .	79
2.1.1	Overview . . . . .	79
2.1.2	Django App steps . . . . .	79
2.1.3	Ember App steps . . . . .	80
2.1.4	Starting up once initial setup is completed . . . . .	81
2.1.5	Previewing a study . . . . .	81
2.1.6	Participating in a study . . . . .	81
2.1.7	Where does my video go? . . . . .	82
2.1.8	Using https . . . . .	82
2.1.9	Further Reading / Useful Links . . . . .	82
2.2	Creating custom frames . . . . .	82
2.2.1	Overview . . . . .	82
2.2.2	Getting Started . . . . .	83
2.2.2.1	A Simple Example . . . . .	83
2.2.2.2	Building out the Example . . . . .	84
2.2.3	Adding CSS styling . . . . .	86
2.2.4	Using mixins . . . . .	87
2.2.4.1	FullScreen . . . . .	87
2.2.4.2	MediaReload . . . . .	87
2.2.4.3	VideoRecord . . . . .	87
2.2.5	Documenting your frame . . . . .	88
2.2.6	Ember debugging . . . . .	88
2.2.7	When should I use actions vs functions? . . . . .	88
2.3	How to capture video in your frame . . . . .	89
2.3.1	Limitations . . . . .	91
2.3.2	How it works . . . . .	91
2.4	Custom randomizer frames . . . . .	92
2.4.1	Overview of ‘choice’ structure . . . . .	92
2.4.2	Making your own . . . . .	92
<b>3</b>	<b>Installation: lookit-api (Django project)</b>	<b>95</b>
3.1	Prerequisites . . . . .	95
3.2	Installation . . . . .	96
3.3	Authentication . . . . .	96
3.4	Handling video . . . . .	96
3.5	Common Issues . . . . .	96
3.6	Continued Installation for developers . . . . .	97
3.6.1	Install Docker . . . . .	97
3.6.2	Install Postgres . . . . .	97
3.6.3	Install RabbitMQ . . . . .	98
3.6.4	Install Ngrok . . . . .	99
3.7	How Do These Programs Work Together? . . . . .	99
<b>4</b>	<b>Installation: ember-lookit-frameplayer (Ember app)</b>	<b>103</b>
4.1	Prerequisites . . . . .	103
4.2	Installation . . . . .	103
4.3	Running / Development . . . . .	104
4.3.1	Code Generators . . . . .	104
4.3.2	Running Tests . . . . .	104
4.3.3	Building . . . . .	104
4.3.4	Writing documentation of frames . . . . .	104
<b>5</b>	<b>Django app implementation notes</b>	<b>105</b>
5.1	Permissions . . . . .	105

5.1.1	Generic best practices . . . . .	105
5.1.2	Guardian, how does it work? . . . . .	105
5.2	Workflow: managing study states . . . . .	106
5.2.1	Why Transitions . . . . .	106
5.2.2	How . . . . .	106
5.2.3	Make a diagram . . . . .	106
5.2.4	Logging . . . . .	106
5.3	Celery tasks . . . . .	106
5.3.1	<code>build_experiment task</code> . . . . .	106
5.3.2	What happens . . . . .	106
5.3.3	<code>build_zipfile_of_videos</code> . . . . .	107
5.3.4	<code>cleanup_builds</code> . . . . .	108
5.3.5	<code>cleanup_docker_images</code> . . . . .	108
5.3.6	<code>cleanup_checkouts</code> . . . . .	108
<b>6</b>	<b>Guidelines for contributors</b>	<b>109</b>
6.1	Prerequisites . . . . .	109
6.2	Getting started . . . . .	110
6.3	Ignoring some files . . . . .	110
6.4	Add your own feature and submit a Pull Request . . . . .	110
6.5	Writing your tests . . . . .	111
6.6	Editing the Lookit documentation . . . . .	112
<b>7</b>	<b>Definitions</b>	<b>113</b>
7.1	Children . . . . .	113
7.2	Demographic Data . . . . .	113
7.3	Experimenter . . . . .	113
7.4	Feedback . . . . .	113
7.5	Groups . . . . .	114
7.6	Organization . . . . .	114
7.7	Organization Site . . . . .	114
7.8	Participants . . . . .	114
7.9	Researchers . . . . .	114
7.10	Responses . . . . .	114
7.11	Study . . . . .	114
<b>8</b>	<b>Technical Glossary</b>	<b>115</b>
8.1	Internal Resources . . . . .	115
8.1.1	Docker . . . . .	115
8.1.2	Postgres . . . . .	117
8.1.3	RabbitMQ . . . . .	117
8.1.4	Ngrok . . . . .	117
8.2	External Resources . . . . .	118
8.2.1	Google Cloud . . . . .	118
8.2.2	Amazon Web Services . . . . .	118
8.2.3	Celery . . . . .	118
8.2.4	Authenticator . . . . .	118
8.2.5	Lookit Ember Frameplayer . . . . .	118
8.2.6	PIPE . . . . .	118
8.2.7	Footnotes . . . . .	119
8.2.8	Endnotes . . . . .	119

The [Lookit codebase](#) contains two main repositories:

- [lookit-api](#), a Django application that houses what were previously separate Experimenter and Lookit applications. Experimenter is a platform for designing and administering research studies, meant for researchers. The Lookit platform is participant-facing, where users can signup and take part in studies.
- [ember-lookit-frameplayer](#), an Ember app that runs studies in the web browser

The documentation you are reading now lives in [lookit-docs](#).

It has been jointly developed by MIT and the [Center for Open Science](#).

Contents:





---

## Using Lookit: for researchers

---

### 1.1 Getting started guide

We are unfortunately not yet able to accommodate most requests to run studies on Lookit! We are focusing on development with the aim of making the platform as easily and broadly usable as possible. However, a limited number of collaborative studies from beta testers are taking place. Here's how to get started if you're working on one of those studies:

0. If you are preparing a new study to run on Lookit, you will first need to get an access agreement signed and start an IRB proposal as described [in the wiki](#).
1. Try out studies at <https://staging-lookit.cos.io> first as a participant to get a feel for what it's like to participate and what some of the standard/existing frames look like. That's the staging server; you won't be confusing anyone who's trying to collect real data.
2. Some background reading:
  - **Please make sure you have actually read the [Terms of Use](#).** There's a bit of legal boilerplate in there, but the bulk is meaningful content about precautions you need to take to protect participant privacy, restrictions on protocols that can be run on Lookit, and what data you (and Lookit) can use and publish. It's important that as the researcher using the platform you actually understand and agree to these terms.
  - Skim through this portion of the documentation ('Using Lookit: for researchers').
  - Skim through [the wiki](#) just so you know what information is available there.
3. Create an outline of how you'd like your study to work, so that we can check the necessary technical functionality exists (or implement things for you) and possibly make you a little mockup to start from (we'll discuss which!). The simplest way to do this is to provide a list of experiment frames, starting from [the sample list](#). Some pieces are fairly standard, like video configuration, consent, or surveys. Most studies, though, will have some sort of "test trials" or experimental procedures where you'll want to carefully think through and write out how things should work.

You can check [the existing frames](#) to see if you think you can just use those. If you're not sure, essentially you'll want to provide enough detail that someone could program your experiment. You may be surprised at how many details you need to think about when you don't have a human running the experiment! If you want

to show images or video on the page, please make a diagram specifying how they are positioned and describing how they are scaled (or not!) depending on screen resolution. Remember that monitors come in different sizes **and shapes**. If you want to play audio or videos, note what triggers them to stop and start, whether they have to complete before something else can happen, etc.

Describe any condition assignment and counterbalancing structure in enough detail that you would be happy to replicate the study if it were someone else's. If what should happen next in the experiment depends on participant responses, make sure you cover all the possible responses.

4. Prepare the content of your study - this is the relatively non-technical (but surprisingly time-consuming) side!
  - [Create all of the stimuli you'll need!](#)
  - Write the text of your study description, prepare a thumbnail image, etc. following the descriptions of each item you'll need to prepare [here](#).
  - Write the text of any study overview, instructions, and debriefing as described [here](#).
5. Implement your study on the staging server: Once a mockup of your study is ready or we've verified that you should be able to implement it yourself, you'll set it up or fine-tune it on the Lookit staging server.
  - Follow [these instructions](#) to get access and log on. Also ask us to get on the Lookit Slack channel for updates!
  - We'll give you access to a study called 'Example study' that you can play around with. Try participating in the original 'Example study' as a participant on the staging server, and check that you can see and approve your consent video in the Consent Manager.
  - Clone the example study so you have your own copy you can edit. Change its title to something like 'Smith getting started study.' Build preview dependencies for your study.
  - Try previewing your study to see what it's like. Make a minor change to the study JSON and preview again to see that you see it.
  - Edit the study to enter your own study's description, thumbnail image, etc. following [these descriptions](#).
  - Now that you have a rough idea of how this all works, read [Building your experiment](#) in more detail (skim the section on randomizers until you're ready to get into that). Add each frame of your planned study, following the [instructions here](#) to try it out as you go.
6. When everything is ready to go, we'll go over day-to-day operation of the study, transfer your study over to the production server, and you'll hit start!

## 1.2 Other helpful resources

- For higher-level info about the Lookit project and running studies on Lookit (e.g., recruitment, state of funding, etc.) please check out (and add to!) [the wiki](#).
- The documentation for individual experiment 'frames' lives here: <https://lookit.github.io/ember-lookit-frameplayer/>
- Running into a problem and want to check if it's a known issue, or have an idea for a handy new feature? Check out and/or add to the issues listed for the [Lookit platform](#) and for the [experiment components/player](#). Or click on 'Milestones' in either repo to take a look at what's coming up in terms of development!

## 1.3 Using the experimenter interface

### 1.3.1 Logging in

Researchers should log into Experimenter via oauth through the Open Science Framework. Visit <https://lookit.mit.edu/exp/> to log in to the production server, or <https://staging-lookit.cos.io/exp/> to log in to the staging server, and click on ‘Open Science Framework’. (Note: if running Lookit locally, you will instead need to authenticate as described in [Setup for custom frame development](#).)

Experimenter

### Login to Experimenter

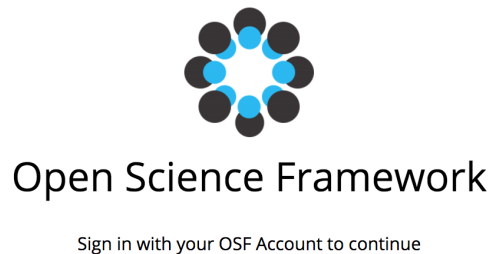
Please sign in to the Open Science Framework or [register](#) for an account and sign in below:

- [Open Science Framework](#)

[Terms of Use](#)

We use regular OSF accounts (you already have one if you have used OSF) for the production server, and staging OSF accounts (you probably don’t have one yet) for the staging server. A staging OSF account is just an account on OSF’s own staging server, where they try out changes ahead of deploying to production.

If you don’t have the appropriate type of OSF account yet, you can register to create one (use the ‘Create Account’ link on the screen shown below). Once you have an account, return to the Lookit experimenter login screen at <https://lookit.mit.edu/exp/> or <https://staging-lookit.cos.io/exp/>, click ‘Open Science Framework’ again, and enter your credentials.



Email:

Password:

**SIGN IN**

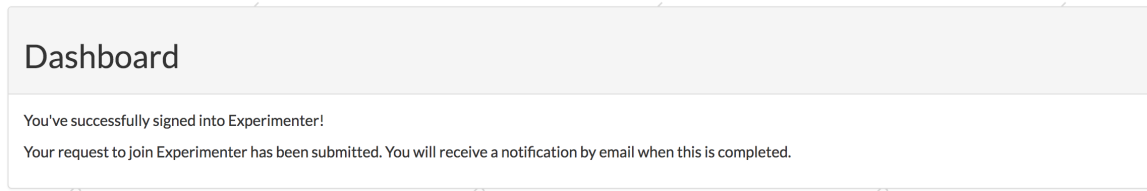
☒ Stay Signed In [Forgot Your Password?](#)

[Login through Your Institution](#) [Back to OSF](#)

[Create Account](#)

If you have not previously logged in to Lookit (or Lookit staging), you should now see a message that ‘Your request to join Experimenter has been submitted. You will receive a notification by email when this is completed.’ **Please tell us**

once you have requested access so we can approve you promptly! New researchers require approval to access Lookit, which is easy (one of us clicking a button) but not automatic.



Otherwise, you will be now be logged into Experimenter.

Researcher accounts can also have children and demographic data, just like participants. If you would like to have a **separate** participant account on Lookit for testing and/or actual participation, use a different email address and sign up for an account using the ‘Sign up’ link on the home page.

## 1.3.2 Managing Studies

### 1.3.2.1 Viewing study list

To view all studies, navigate to `/exp/studies/`. (We will use this short format to indicate relative paths starting with the Lookit site you are using - e.g., <https://lookit.mit.edu/exp/studies/> or <https://staging-lookit.cos.io/exp/studies/>). A researcher must have been added to an organization to view this page. From there, the researcher can only see studies they have permission to view. Org admins and org reads can see all studies that belong to their organization. If the user is a basic researcher, they can only view studies which they have created or to which they have been explicitly added.

You can filter studies by name or by keywords in the description. Additionally, you can sort on various study states like “Created” or “Submitted”, or filter on your own studies by selecting “My Studies”. You can also sort on study name, study end date, and study begin date.

See Particip

## Manage Studies

+ Create Study

Active Submitted Approved Created Deactivated MyStudies All

Name ^v Begin Date ^v End Date ^v

<a href="#">Baby laughter games</a> In this study, you and your baby will perform a series of short games, including the crowd-favourite, "Peekaboo"! Sometimes repetition is what makes something funny, so we will ask that you perform each game three times. Study Creator: Kim Scott Status: Active	Mar 22, 2019 N/A	Completed Responses: 2 Incomplete Responses: None Approved Consent: 1 Pending Judgement: 1
<a href="#">Basic WebRTC recording test study</a> This is a "study" to check that the new webRTC recorder is working. Study Creator: Kim Scott Status: Active	Mar 21, 2019 N/A	Completed Responses: 1 Incomplete Responses: 1 Approved Consent: 1 Pending Judgement: 1
<a href="#">Copy of Example study</a> Here is an example study demonstrating a simple sequence of experiment frames. Study Creator: Se-Woong Park Status: Created	N/A N/A	Completed Responses: None Incomplete Responses: None Approved Consent: None Pending Judgement: None

### 1.3.2.2 Creating a study

To create a study, navigate to `/exp/studies/create/`. You'll need to provide values for the fields as described in [Setting study fields](#).

[Manage Studies](#) / [Create Study](#)

Create Study

Name

Image

Choose File
No file chosen

Please keep your file size less than 1 MB

Short Description

Give your study a description here.

Purpose

Explain the purpose of your study here.

Exit URL

Specify the page where you want to send your participants after they've completed the study.

Participant Eligibility

Minimum Age Cutoff

Year(s)
Month(s)

Maximum Age Cutoff

Year(s)
Month(s)

Duration

Researcher/Contact Information

☐ Discoverable - Do you want this study to be publicly discoverable on Lookit once activated?

Build Study - Add JSON

```
{ "frames": {}, "sequence": [] }
```

Add the frames of your study as well as the sequence of those frames. This can be added later.

Study Type

Specify the build process as well as the parameters needed by the experiment builder. If you don't know what this is, just select the default.

Cancel
+ Create Study

[Terms of Use](#)

### 1.3.2.3 Study detail page

To view a single study, click on it from the study list. A researcher must have permission to view this study specifically. Org admins and org reads can view all studies in their organization. A basic researcher can only view this study if they have been explicitly added as a study admin or study read. At the top, you see many of the study details that you

entered when you created the study. The UUID is also displayed; this is your study's unique identifier and is used in the direct link to the study.

At the top right, you have options to edit the study, view responses, email participants, or clone the study. Cloning will create a copy of the study but add the logged in user as the creator. The clone will be moved back into "Created" status (e.g., if the current study is actively collecting data, the cloned study will not be - it will need to be approved before it can be started). Study logs of when the study changed state are at the bottom of the page.

The only things that can be edited from this page are the study status and researcher list. The current study status is displayed, as well as a dropdown with the available states. Only users that have permission to edit the study state can make these changes, meaning organization admins, or study admins. The available states where you can move the study depend on what state is next in the sequence, as well as your current level of permissions. For example, if a study's current state is "Created", that study can only be "Submitted" for review, or "Archived", which removes the study from display. Comments can only be left on the study if it is being rejected. Only organization admins can approve or reject a study.

See Partic

Manage Studies / Example study

## Example study



Last edited: Apr 04, 2019

[Edit Study](#) [View Responses](#) [Take Action -](#)

Here is an example study demonstrating a simple sequence of experiment frames.

### Purpose:

This is for new researchers to play with! You will be given "read" access to this study, so you will be able to see it (and its data) but not make changes or approve consent videos. You should go to "Take action -> Clone study" to make your own copy, which you can then edit to your heart's content :)

Duration: 10 minutes Exit URL: <https://staging-lookit.cos.io/>

Participant eligibility: For babies ages 4 to 8 months Compensation:

Minimum age cutoff: 0 years 3 months 27 days Maximum age cutoff: 0 years 8 months 4 days

UUID: 87b437cd-0cc8-49a0-b175-6a51ed1bd5d5

Discoverability: Public. Your study is active and public. Participants can access it at your study link, and it can be found listed in the [study listing page](#).

### Study link:

<https://staging-lookit.cos.io/studies/87b437cd-0cc8-49a0-b175-6a51ed1bd5d5>

Change State -

Example study is currently active.

Study is collecting data

Build Dependencies ↗

Study dependencies built.

Build Preview Dependencies 👁

Preview dependencies built.

### Manage Researchers

Search organization



### Researchers

Researchers belonging to this study's admin and read groups. MIT Admins will automatically be able to edit this study, regardless of study group.

Name	Permissions	
Laurie Bayet	<a href="#">Read</a>	
Annie Cardinaux	<a href="#">Read</a>	
Junyi Chu	<a href="#">Read</a>	
Gleb Iakovlev	<a href="#">Read</a>	
Bria Long	<a href="#">Read</a>	
Stephan Meylan	<a href="#">Read</a>	
Se-Woong Park	<a href="#">Read</a>	
Kim Scott * MIT Admin	Admin	
Goldsmiths Testing	<a href="#">Read</a>	

### Study Logs:

April 4, 2019, 3:38 p.m.	Example study study activated by Kim Scott.
April 4, 2019, 3:38 p.m.	Example study study approved by Kim Scott.
April 4, 2019, 3:38 p.m.	Example study study submitted by Kim Scott.
April 3, 2019, 7:49 p.m.	Example study study dependencies built by Rico Rodriguez.



### 1.3.2.4 Study status

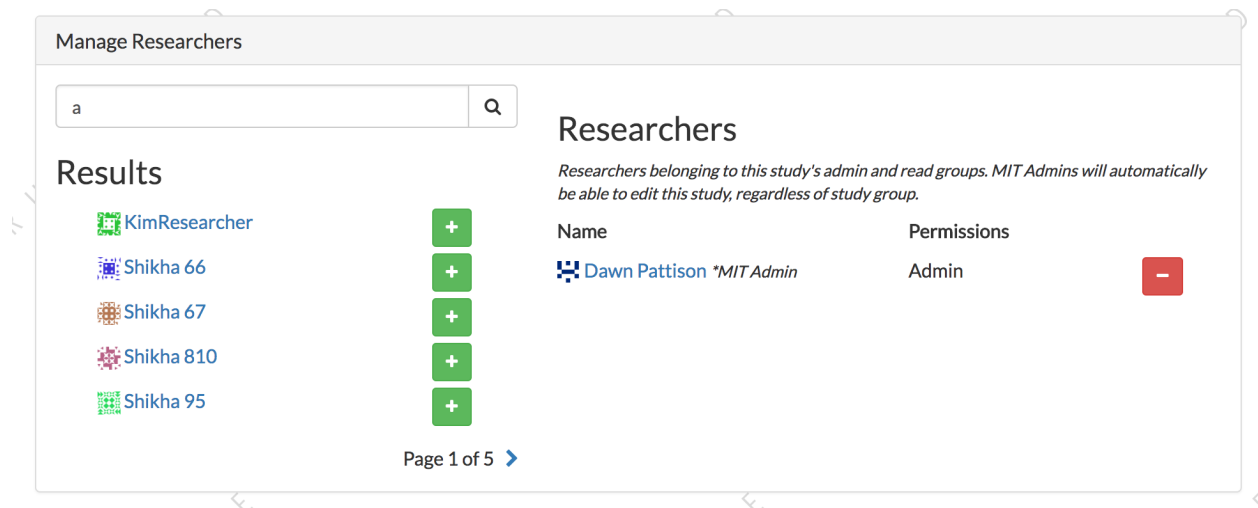
New studies must be submitted and approved by Lookit before they can be started. Once approved, researchers with study admin permissions can independently start/pause data collection at will; however, if any changes are made to the study it will be automatically rejected and will require re-approval. The study approval process is intended to give Lookit staff an opportunity to check that studies comply with the Terms of Use and to provide support if necessary. Researchers will receive email notifications when their study is approved or rejected.

The possible study states are:

- *created*: Study has been initially created, but has not been submitted for approval
- *submitted*: Study is submitted and awaiting approval by an organization admin
- *approved*: Study has been approved by an organization admin to run on Lookit, but is not yet active
- *rejected*: The study has been rejected by an organization admin. The study should be edited before resubmitting.
- *active*: Study is active and can be collecting data. If the study is also marked “Discoverable”, the study will show up on Lookit’s study list.
- *paused*: Study is not actively collecting data or visible on Lookit
- *deactivated*: Study is done collecting data
- *archived*: Study has been archived and removed from search

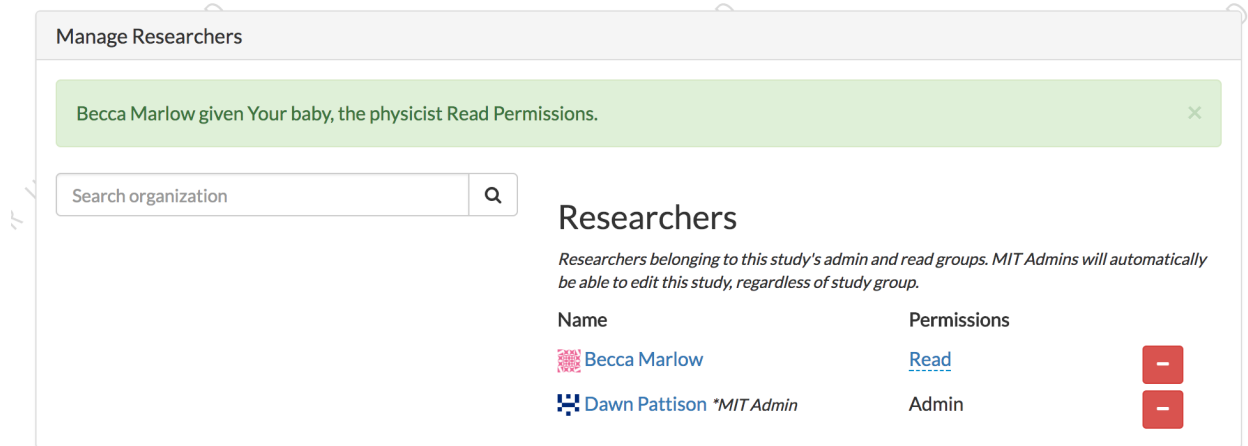
### 1.3.2.5 Adding researchers to your study

Halfway down the study detail page, you can see the researchers that have study admin or study read permissions to your study. In the search box, you can look for an existing Lookit researcher (this must be someone who has already been added to your organization).



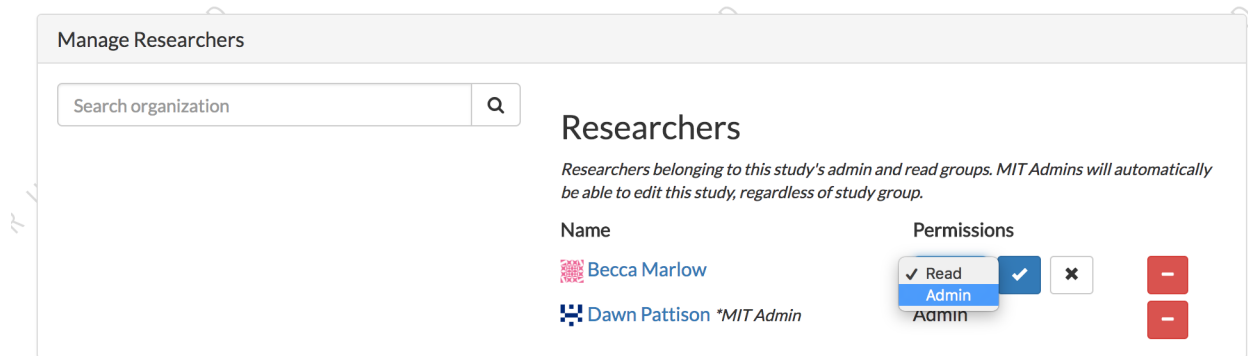
Click the green plus to add them to your study. They are given study read permissions by default; this allows them to see all study details and participant data and to approve consent videos, but not to change study details, change study status (e.g. start/stop data collection), or add other researchers.

If the researcher you are adding happens to also be an organization admin, they will have admin permissions on your study. These researchers that are also org admins are denoted by an asterisk, followed by the <name of your organization>-Admin.



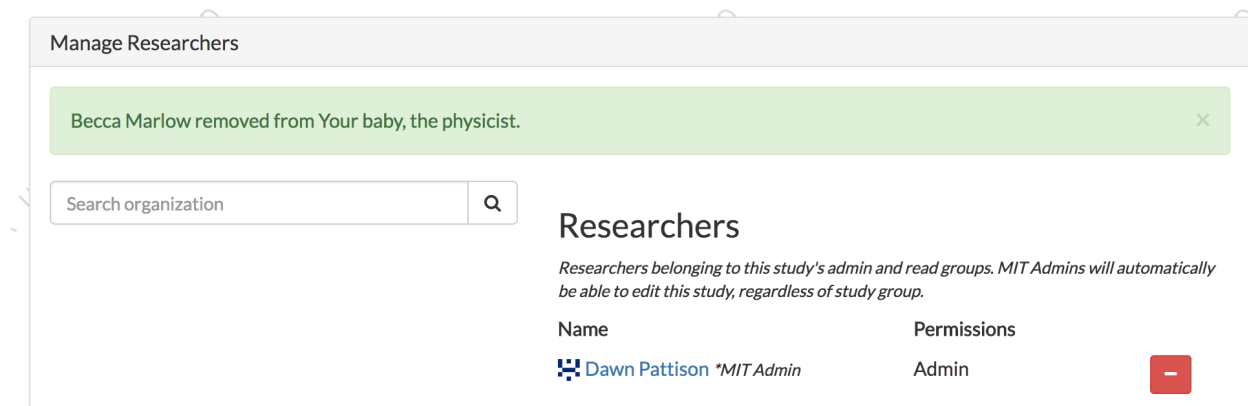
### 1.3.2.6 Editing researcher permissions on a study

To edit a researcher, select read or admin permissions in the dropdown beside the researcher name and click the checkmark. This will automatically give the researcher read or admin permissions. There must be at least one study admin at all times.



### 1.3.2.7 Deleting researcher permissions

To remove a researcher from a study, click the red minus button beside the researcher's name. This will automatically remove the user's study admin or study read permissions. There must be at least one study admin at all times, so it's possible that you won't be able to remove a researcher.



### 1.3.2.8 Study edit page

On the study edit page, you can update much of the metadata about the study. You can only view this page if you have permission to edit this particular study, meaning org admins or study admins. At the top of the page, you can edit fields like Name, and Description. See [Creating a Study](#) for more details.

To edit fields, change the information and click Save Changes in the middle of the page. If your study has already been approved, then the save button will be red. Otherwise it will be green. If your study has already been approved, then editing key details will automatically put the study in a rejected state. You must resubmit your study and get it approved again by an organization admin to run the study on the Lookit platform.

At the bottom of the edit study page, you can make edits to your study's structure (the frames, or pages, in your experiment), and the sequence of those frames. You can also make advanced edits to the commits we are using to build your study.

Experimenter

See ParticipantsManage OrganizationManage StudiesDawn Pattison

Manage Studies / Your baby, the physicist / Edit

Description and Discoverability

Name

Your baby, the physicist

Image

Currently: [study\\_images/baby.png](#)  
Change:  

Choose FileNo file chosen

Please keep your file size less than 1 MB

Short Description

Your baby watches pairs of short video clips of physical events. In each pair, something pretty normal for our world happens on one side: e.g., a ball rolls off a table and falls to the round. On the other side, something different happens: e.g., the ball rolls off a table and falls UP!

Give your study a description here.

Purpose

Where your baby chooses to look can tell us about his or her expectations of how the physical world works. Although your baby isn't ready to study physics, he or she is already learning the basics: Should things fall up or down or not at all? Should they keep going once they start moving?

Explain the purpose of your study here.

Exit URL

<https://staging-lookit.cos.io>

Specify the page where you want to send your participants after they've completed the study.

Participant Eligibility

For children between 4 and 12 months old at start of study

Minimum Age Cutoff

Year(s)

Month(s)

0

4

Maximum Age Cutoff

Year(s)

Month(s)

1

3

Duration

Fifteen minutes

Researcher/Contact Information

[pattison.dawn@cos.io](mailto:pattison.dawn@cos.io)

☒ Discoverable - Do you want this study to be publicly discoverable on Lookit once activated?

Cancel

Save Changes

Manage Researchers

Search organization

Q

Researchers

Researchers belonging to this study's admin and read groups. MIT Admins will automatically be able to edit this study, regardless of study group.

Name	Permissions
Dawn Pattison *MIT Admin	Admin

Build Study

Add/Modify study components

Status: Active

Change status ...

Comments:

You can only leave comments when you are an admin of this study.

14

Chapter 1. Using Lookit: for researchers

### 1.3.2.9 Editing study structure

For more information about how to specify what happens during your study, see [Building an Experiment](#).

To edit a study's structure, click 'Edit study' from the study detail page. You must be a study admin or org admin to view this page. From this 'study edit' page, you can edit the study's structure and the study type. The study structure specifies the frames (or pages) of your experiment, and also specifies the sequence.

To edit the structure, click on the JSON block. A JSON editor will appear. Click on “Beautify” in the top right corner for better readability. Note that any invalid JSON will be shown via a little red X at the left of the relevant line! Once you are happy with your changes click ‘Close’. Then hit “Save” in the bottom right corner. If your study has already been approved, then clicking “Save” will automatically reject the study. You will have to resubmit it for an organization admin to reapprove.

```

1- {
2-   "frames": {
3-     "exit-survey": {
4-       "id": "exit-survey",
5-       "kind": "exp-exit-survey",
6-       "title1": "Almost done!",
7-       "title2": "Thank you! You're all done.",
8-       "exitMessage": "",
9-       "exitThankYou": "Thank you so much for your help! We appreciate and learn from every video we receive in the lab (even if what we learn is that your
10-         kiddo thinks this study is boring and we need to up our game.)",
11-       "idealSessionsCompleted": 15,
12-       "idealDaysSessionsCompleted": 60
13-     },
14-     "mood-survey": {
15-       "id": "mood-survey",
16-       "kind": "exp-mood-questionnaire"
17-     },
18-     "instructions": {
19-       "id": "instructions",
20-       "kind": "exp-physics-intro"
21-     },
22-     "video-config": {
23-       "id": "video-config",
24-       "kind": "exp-video-config",
25-       "instructions": "Make sure your camera is working and you can see yourself below! Important: you'll need to check 'Remember' when you allow access, so
26-         that it'll still work on the next screen."
27-     },
28-     "video-consent": {
29-       "id": "video-consent",
30-       "kind": "exp-video-consent",
31-       "blocks": [
32-         {
33-           "text": "Observing your child's behavior during this experimental session will help us to understand how infants and children use evidence to
34-             learn and make predictions about the world.",
35-           "title": "About the study"
36-         },
37-         {
38-           "text": "Your and your child's participation in this session are completely voluntary. If you and your child choose to participate, you may stop
39-             the session at any point with no penalty. Please pause or stop the session if your child becomes very fussy or does not want to participate.
40-             If this is a study with multiple sessions, there are no penalties for not completing all sessions.",
41-           "title": "Participation"
42-         }
43-       ]
44-     }
45-   }
46- }

```

To preview your study, click “Try Experiment”. (You will need to build preview dependencies first if you haven’t yet, or if you’ve changed the study type or which code to use.)

### 1.3.2.10 Editing study type

To edit a study’s type, click ‘Edit study’ from the study detail page.

The study type is the application you’re using to enable participants to take a study. Right now, we just have one option, the [Ember Frame Player](#). It’s an ember app that can talk to our API. All the frames in the experiment are defined in `ember-lookit-frameplayer`, and the `exp-player` component can cycle through these frames.

**If you don’t want any customization and want to use the existing player and frames, just select the defaults.** These are advanced options!

What does each field mean?

- The `player_repo_url` is the repo where the frames and the player are stored. This is the default `player_repo_url`: <https://github.com/lookit/ember-lookit-frameplayer>. Advanced users may want to define their own custom frames for use with Lookit studies beyond those provided in the core library. (For more information about how to do this, see <https://lookit.readthedocs.io/en/develop/developing-frames.html>.) To use your own frame definitions, set the `addons_repo_url` to your own fork of the `ember-lookit-frameplayer` repo (e.g., <https://github.com/yourname/ember-lookit-frameplayer> instead of <https://github.com/lookit/ember-lookit-frameplayer>). You can then choose any commit SHA from your own repo.
- The `last_known_player_sha` is the commit of the player repo to use. If you don’t add this, it will point to the latest commit in the default branch. To browse commits available for the experiment player `ember-lookit-frameplayer` and see what might have changed, you can look through <https://github.com/lookit/ember-lookit-frameplayer/commits/>. (Previously, `ember-lookit-frameplayer` contained just the frame player, with frames defined separately in an `exp-addons` repo. Do not use a frameplayer SHA from before these two repos were merged together, in March 2019)

Leave the field `last_known_player_sha` blank to use the default - the latest versions of the experiment player and frames that Lookit provides. When you build dependencies, the commit SHAs (unique IDs) of the latest versions will be fetched and filled in, so that you will continue to use this version for your experiment unless you deliberately update.

**Important:** Whenever you update the code versions you are using, you will need to build dependencies again to preview and to activate your study. This build process creates a special environment just for your study using exactly the code you selected, so that your study will continue to run as you designed it. By storing builds on Google Cloud Storage, pointing to specific commits, we can keep edits to frames from unintentionally breaking another study. You only need to build dependencies when you have changed the commit SHAs here - not when you update your study JSON or other data like the age range.

### 1.3.2.11 Viewing individual study responses

For information about interpreting study responses, see [Experiment data](#).

To view a study's response, navigate to your study and click 'View Responses,' then 'Individual responses'. You must have permission to view this study's responses, which means you must be an Organization Admin, Organization Read, or belong to the Study Admin or Study Read groups.

Responses only show up in this view once you have confirmed that the participant provided informed consent to participate using the Consent Manager.

On the left, you have a list of participants that have responded to your study, with the response id, the study's completion status, and the date it was modified. When you click on a participant, the JSON of that participant's response is shown on the right. You can download the individual participant's JSON response by clicking "Download Individual Response JSON". Alternatively, you can select CSV in the dropdown, and click "Download Individual Response CSV".

Beneath the CSV/JSON response data are any individual video attachments that are linked to that participant's response. Exception: if the participant selected the 'withdraw video' option in an exit-survey frame at the end of the study, all video except for the consent video is unavailable (and will be deleted from Lookit servers as well in 7 days). There is a potential rare edge case where you access video while the participant is still doing the study, and then they withdraw, so you should still verify that none of your participants have withdrawn video.

Manage Studies / Example study / Individual Responses

## Individual Responses

Consent Manager Individual Responses All Responses Demographic Snapshots Attachments

User ID ^v	Response ID	Status ^v	Date ^v
62	579	Complete	4/02/2019
64	578	Complete	4/02/2019
64	577	Complete	4/01/2019
62	575	Complete	4/01/2019
64	576	Complete	4/01/2019
62	573	Complete	4/01/2019
62	567	Complete	3/21/2019

Page 1 of 1

JSON

Download Individual Response JSON

```
{
  "response": {
    "id": 579,
    "uuid": "d66a5e7f-58fb-4052-823d-beb67d",
    "sequence": [
      "0-video-config",
      "1-video-consent",
      "2-instructions",
      "3-video-preview-exp",
      "4-video-preview",
      "5-example-survey"
    ]
  }
}
```

### Attachments

videoStream\_87b437cd-0cc8-49a0-b175-6a51ed1bd5d5\_1-video-consent\_d66a5e7f-58fb-4052-823d-beb67d26e5ed\_1554240738663\_373.mp4 [Download](#)

videoStream\_87b437cd-0cc8-49a0-b175-6a51ed1bd5d5\_9-pref-phys-videos\_d66a5e7f-58fb-4052-823d-beb67d26e5ed\_1554240829794\_783.mp4 [Download](#)

[Terms of Use](#) | [Privacy](#)

### 1.3.2.12 Viewing all study responses

To view all of the responses to a study with confirmed consent, click 'View Responses' from the study detail page and then click 'All Responses.' You must have permission to view this study's responses, which means you must be an Organization Admin, Organization Read, or belong to the Study Admin or Study Read groups.

By default, all study responses are displayed in JSON format. To download as CSV, select CSV in the dropdown and download. The study response data is supplemented with the study id, participant ids and nickname, and the associated child info.

Manage Studies / Example study / Demographic Snapshots

## All Responses

Consent Manager Individual Responses All Responses Demographic Snapshots Attachments

JSON

Download all 7 demographic snapshots as JSON

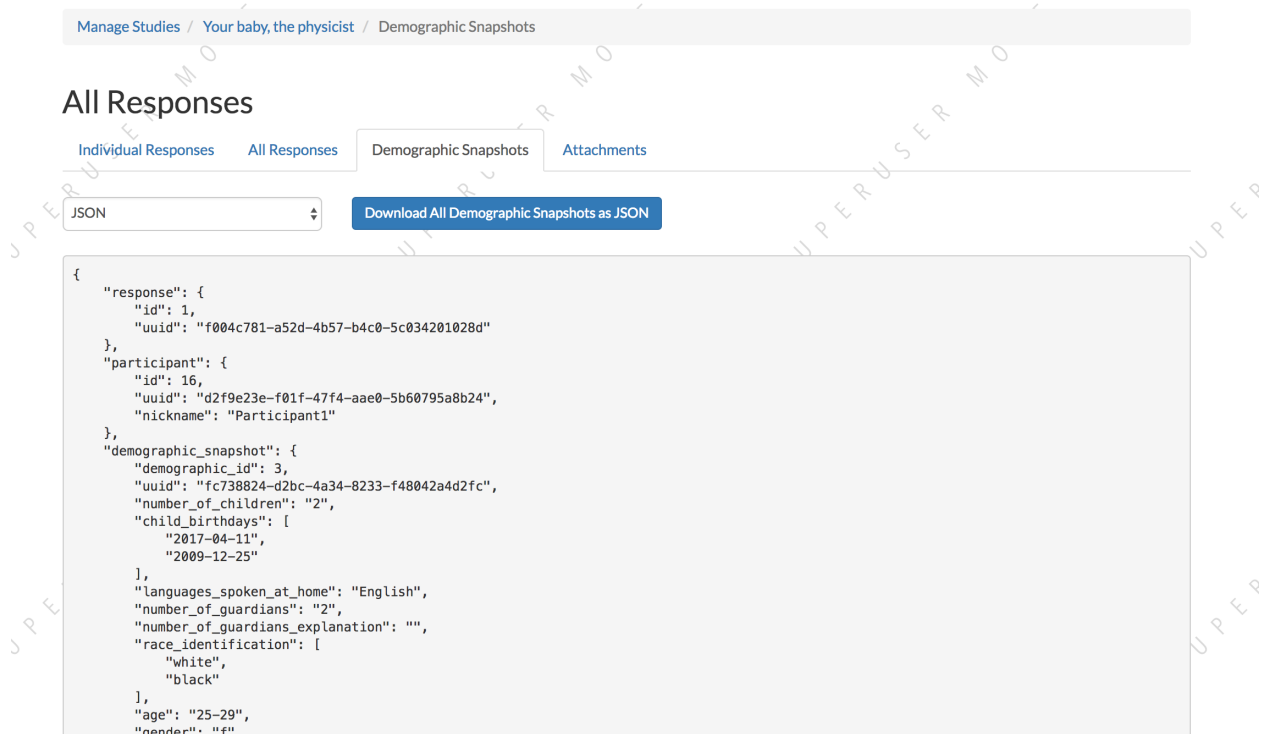
[Terms of Use](#) | [Privacy](#)



### 1.3.2.13 Viewing demographics of study participants

To view the demographics of participants that have responded to your study and have confirmed consent, click ‘View Responses’ from the study detail page and then click ‘Demographic Snapshots.’ You must have permission to view this study’s responses, which means you must be an Organization Admin, Organization Read, or belong to the Study Admin or Study Read groups.

This list is generated by looping through all the responses to your study, and displaying the demographics of the associated participant. If a participant has responded multiple times, the demographics will appear multiple times. Demographic data was versioned, so the demographics associated with each response will be the demographics that were current at the time the participant responded to the study. You can download the demographics in JSON or CSV format.



```
{
  "response": {
    "id": 1,
    "uuid": "f004c781-a52d-4b57-b4c0-5c034201028d"
  },
  "participant": {
    "id": 16,
    "uuid": "d2f9e23e-f01f-47f4-aae0-5b60795a8b24",
    "nickname": "Participant1"
  },
  "demographic_snapshot": {
    "demographic_id": 3,
    "uuid": "fc738824-d2bc-4a34-8233-f48042a4d2fc",
    "number_of_children": "2",
    "child_birthdays": [
      "2017-04-11",
      "2009-12-25"
    ],
    "languages_spoken_at_home": "English",
    "number_of_guardians": "2",
    "number_of_guardians_explanation": "",
    "race_identification": [
      "white",
      "black"
    ],
    "age": "25-29",
    "gender": "f"
  }
}
```

### 1.3.2.14 Viewing all study videos

To view all video responses to your study from sessions with confirmed consent, click ‘View Responses’ from the study detail page and then click ‘Attachments.’ You can filter on video attachment name. The format of the video names is *videoStream\_{study\_uuid}\_{order-frame\_name}\_{response\_uuid}\_{timestamp}\_{randomDigits}.mp4*

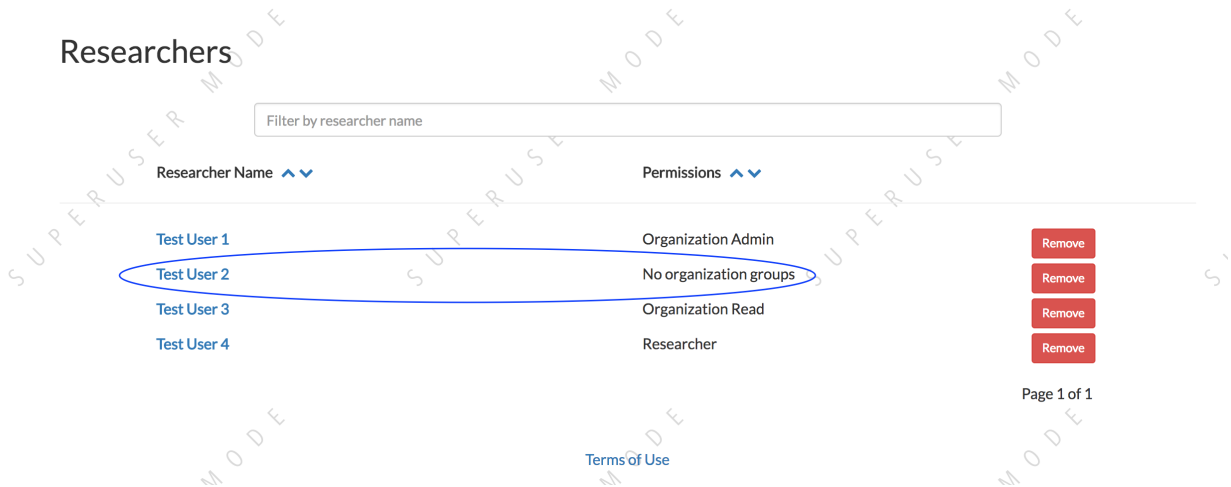
Video attachments can be downloaded individually. You also have the option of bulk downloading all consent videos for your study, or bulk downloading all responses. The bulk download will take place asynchronously, so once the videos have been downloaded and put in a zip file, you will get an email telling you this is done.

## 1.3.3 Managing your Organization

Currently all researchers using Lookit are part of a single ‘MIT’ organization. The organization construct will eventually allow labs to manage access for their own students and RAs. For now, though, these instructions just apply to Lookit admins.

### 1.3.3.1 Adding researchers to your organization

Navigate to *Manage Organization* <https://lookit.mit.edu/exp/researchers/>. Only users with organization admin and organization read permissions can view other researchers in the org. The researchers displayed are researchers that currently belong to your organization, or researchers still needing approval. Researchers awaiting approval have “No organization groups” listed as the permission. Navigate to a researcher awaiting approval (only organization admins are permitted to do this).

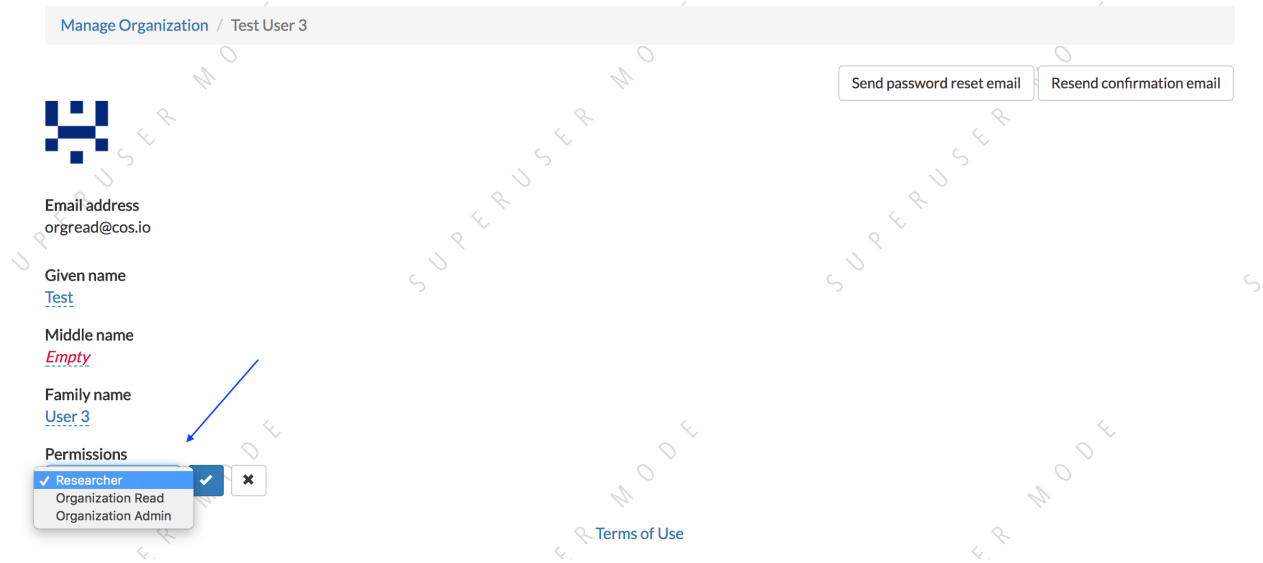


Under permissions at the bottom of the researcher detail page, select *Researcher*, *Organization Read*, or *Organization Admin* from the dropdown, and click the check mark. This will give that researcher the associated permissions and add them to your organization. They will receive an email notification.



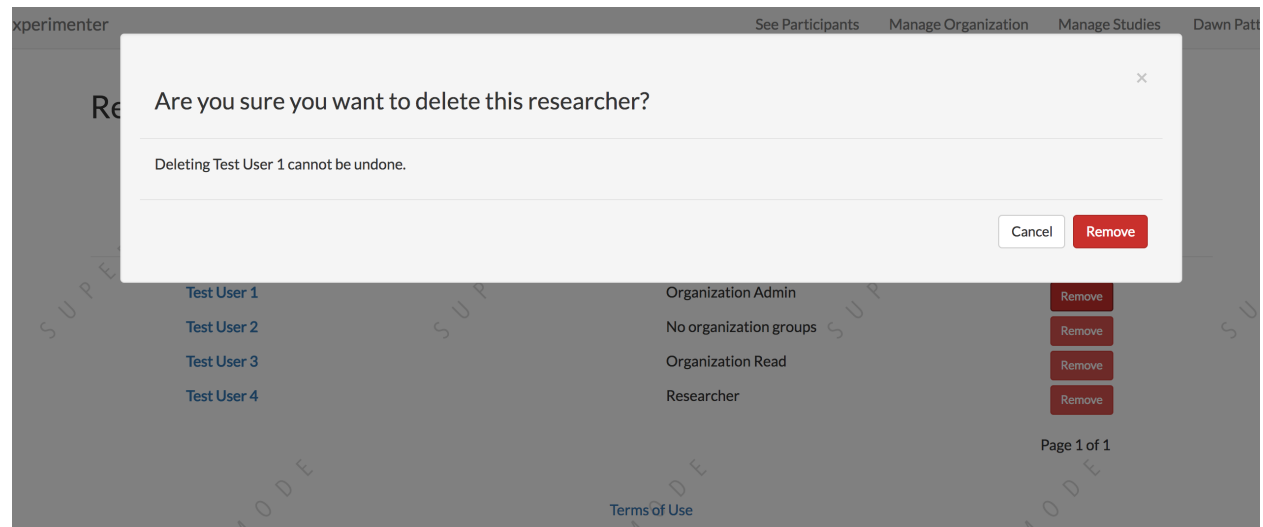
### 1.3.3.2 Editing a researcher's organization permissions

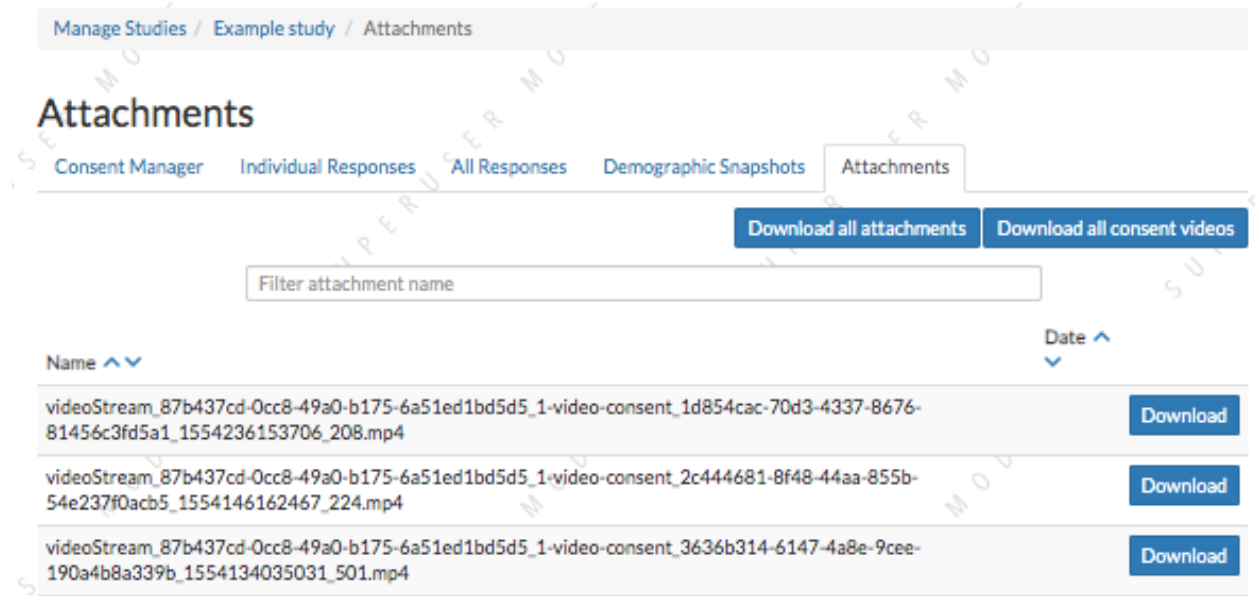
Navigate to a particular researcher's detail page [https://lookit.mit.edu/exp/researchers/<researcher\\_id>](https://lookit.mit.edu/exp/researchers/<researcher_id>). Only organization admins can view this page. Under permissions at the bottom of the researcher detail page, select *Researcher*, *Organization Read*, or *Organization Admin* from the dropdown, and click the check mark. This will modify the researcher's permissions.



### 1.3.3.3 Deleting a researcher's organization permissions

Navigate to *Manage Organization* <https://lookit.mit.edu/exp/researchers/>. Only users with organization admin and organization read permissions can view other researchers in the org. Click “Remove” beside the researcher you wish to delete, and then click “Remove” again in the confirmation modal. The researcher will be marked as inactive and will no longer be permitted to login to Experimenter.





## 1.4 Setting study fields

When creating or editing a study, you can set the value of the following fields. Below is more information about each:

### 1.4.1 Name

Participant-facing title of your study; must be <255 characters. Shoot for a short, catchy title; depending on how you advertise your study, you may want participants to be able to recognize and select it from the studies page. If you plan on running similar follow-up studies and want them to be easily distinguishable, avoid titles that encompass your entire research program like “Infant Language Study.”

### 1.4.2 Image

Thumbnail image that will be displayed to participants on Lookit’s studies page. File must be an image-type, and please keep the file size reasonable (<1 MB). Sometimes your stimuli are a good basis for creating this image, or it can be something that conceptually represents your study or shows what it looks like to participate.

### 1.4.3 Short description

Describe what happens during your study here (1-3 sentences). This should give families a concrete idea of what they will be doing - e.g., reading a story together and answering questions, watching a short video, playing a game about numbers.

### 1.4.4 Purpose

Explain the purpose of your study here (1-3 sentences). This should address what question this study answers AND why that is an interesting or important question, in layperson-friendly terms. Note: this tends to be harder than you’d think - it’s not just you! Imagine all the time you spend getting comfortable explaining the point of a study in the lab (or training RAs on the same), distilled into this task. Plus you don’t get to interact with the parent to gauge their

interest level or familiarity first. Take your time and read this out loud as you work. Some things to check: Is it too specific - is a reasonable response “okay, you will find out whether X is true, but why does that matter?” Is it too general - could you write the same thing about a follow-up study you’re planning or another study going on in your lab?

### 1.4.5 Compensation

Provide a description of any compensation for participation, including when and how participants will receive it and any limitations or eligibility criteria (e.g., only one gift card per participant, being in age range for study, child being visible in consent video). Please see the [Terms of Use](#) for details on allowable compensation and restrictions. If this field is left blank (which is okay if you’re not providing compensation beyond the joy of participation) it will not be displayed to participants.

### 1.4.6 Exit URL

Must enter a URL. After the participant has completed the study, they will be automatically redirected to the exit URL. Typically this is just <https://lookit.mit.edu/>

### 1.4.7 Participant eligibility description

Freeform participant-facing eligibility string, of the form ‘For...’ (e.g., ‘For babies under 1 year old’). Make this readable so participants understand if their child can take part in the study.

This is **not** directly used to automatically check eligibility, so you can include criteria that may not yet be possible to check for automatically - e.g., this study is for girls whose favorite color is orange.

Age limits specified here should be carefully considered with respect to the *minimum and maximum age cutoffs* which **are** used for automatic verification of eligibility.

### 1.4.8 Criteria expression

Providing this expression allows you to specify more detailed eligibility criteria for your study than a single age range. When a parent selects a child to participate in a study, he or she will see a warning under any of the following conditions:

- The child is under the minimum age specified (see *minimum and maximum age cutoffs*)
- The child is over the maximum age specified (see *minimum and maximum age cutoffs*)
- The child is within the specified age range, but doesn’t meet the eligibility criteria defined in this expression

Note that while a warning is displayed, ineligible participants are not actually prevented from participating; this is deliberate, to remove any motivation for a curious parent to fudge the details to see what the study is like.

You may want to use the criteria expression to specify additional eligibility criteria beyond an age range - for instance, if your study is for a special population like kids with ASD or bilingual kids. You do **not** need to specify your age range here in general; participant eligibility checks will require the child meet the *minimum and maximum age cutoffs* AND these criteria.

Every child in the Lookit database has a number of fields associated with it, ranging from gestational age to languages spoken in the home, which can be used in determining eligibility. In the study edit and create views, you can formulate your criteria expression as a boolean expression with embedded relational expressions, using a domain specific query language.

You can put together your expressions using the query fields below; the operators *AND*, *OR*, *NOT*, *<*, *<=*, *=*, *>*, and *>=*; and parentheses. If your expression is invalid you will see an error when you try to save your study.

### 1.4.8.1 Query fields

Query Handle	Value Type	Examples	Notes
[ <i>CONDITIONS</i> ]	N/A	deaf, hearing_impairment, NOT multiple_birth	See below for full list of available options.
speaks_[ <i>LANGUAGE</i> ]	N/A	speaks_en, NOT speaks_ja, speaks_ru	See below for full list of available options.
gestational_age_in_weeks	integer	gestational_age_in_weeks <= 40, gestational_age_in_weeks = na	Values are 23 through 40 and na
gender	string	gender = f, gender != o	Male (m), Female (f), Other (o), or Not Available (na).
age_in_days	integer	age_in_days <= 1095, age_in_days > 365	

### 1.4.8.2 Criteria expression examples

**Deaf children only** deaf

**Multiple-birth children who are either under 1 year old or over 3 years old** multiple\_birth AND (age\_in\_days >= 1095 OR age\_in\_days <= 365)

**Girls who are exposed to both English and Spanish** gender = f AND speaks\_en AND speaks\_es

**Children born late preterm whose adjusted age is about 6 weeks** (gestational\_age\_in\_weeks = 34 AND (age\_in\_days >= 72 AND age\_in\_days < 102)) OR (gestational\_age\_in\_weeks = 35 AND (age\_in\_days >= 65 AND age\_in\_days < 95)) OR (gestational\_age\_in\_weeks = 36 AND (age\_in\_days >= 58 AND age\_in\_days < 88))

### 1.4.8.3 Characteristics and conditions

Query Handle	Condition/Characteristic
autism_spectrum_disorder	Autism Spectrum Disorder
deaf	Deaf
hearing_impairment	Hearing Impairment
dyslexia	Dyslexia
multiple_birth	Multiple Birth (twin, triplet, or higher order)

### 1.4.8.4 Language codes

Code	Language
en	English
am	Amharic

Continued on next page

Table 1 – continued from previous page

Code	Language
bn	Bengali
bho	Bhojpuri
my	Burmese
ceb	Cebuano
hne	Chhattisgarhi
nl	Dutch
egy	Egyptian Spoken Arabic
fr	French
gan	Gan
de	German
gu	Gujarati
hak	Hakka
ha	Hausa
hi	Hindi
ig	Igbo
id	Indonesian
pes	Iranian Persian
it	Italian
ja	Japanese
jv	Javanese
cjy	Jinyu
kn	Kannada
km	Khmer
ko	Korean
mag	Magahi
mai	Maithili
ms	Malay
ml	Malayalam
cmn	Mandarin
mr	Marathi
nan	Min Nan
mor	Moroccan Spoken Arabic
pbu	Northern Pashto
uzn	Northern Uzbek
or	Odia
pl	Polish
pt	Portuguese
ro	Romanian
ru	Russian
skr	Saraiki
sd	Sindhi
so	Somali
es	Spanish
su	Sunda
tl	Tagalog
ta	Tamil
te	Telugu
th	Thai
tr	Turkish

Continued on next page

Table 1 – continued from previous page

Code	Language
uk	Ukrainian
ur	Urdu
vi	Vietnamese
lah	Western Punjabi
wuu	Wu
hsn	Xiang Chinese
yo	Yoruba
yue	Yue

### 1.4.9 Minimum and maximum age cutoffs

Integer fields specifying minimum/maximum ages of participants (inclusive). Eligibility is calculated based on the child's current age in days; this is compared to the minimum/maximum ages in days, calculated as  $365 \times \text{years} + 30 \times \text{months} + \text{days}$ . Participants under the age range see a warning indicating that their data may not be used, and suggesting that they wait until they're in the age range. Participants over the age range just see a warning indicating that their data may not be used. Participants are never actually prevented from starting the study, to remove motivation for a curious parent to fudge the child's age.

Note that these ages do **not** in all cases correspond exactly to the child's age in 'calendar months' or 'calendar years' (e.g., 'one month' if that month is February). In general, you want to avoid a situation where the parent thinks their child should be eligible based on the participant eligibility string (e.g., "my child is one month old, she was born February 3rd and it's March 4th!") but sees a warning when trying to participate. You can do this by narrowing the eligibility criteria in the freeform string and/or by expanding them in the cutoffs here. If one has to align better with your actual inclusion criteria, in general you want that to be the minimum/maximum age cutoffs.

### 1.4.10 Duration

Approximately how long does it take to do your study, start to finish? (Try it if you're not sure; include time to read the instructions.) You can give an estimate or range.

### 1.4.11 Researcher contact information

This should give the name of the PI for your study, and an email address where the PI or study staff can be reached with questions. Format: PIs Name (contact: [youremail@lab.edu](mailto:youremail@lab.edu)). This is displayed to participants on the study detail page before they choose to participate, as well as substituted into your consent form and exit survey, so in general the name needs to be the person who's listed as PI on your IRB protocol (although it may not need to be their personal email address).

### 1.4.12 Discoverable

Do you want this study to be listed on the Lookit studies page when it's active? Check this box to list the study there. If the box is unchecked, the study will be 'non-discoverable' and participants will only be able to get to it by following a direct link with your study ID. This may be helpful if, for instance, you want to run a follow-up study (with in-lab on online participants) and want to send the link to a limited number of people, or if your inclusion criteria are very limited (e.g., a rare genetic disorder) and you want to recruit specifically without getting any random curious families stopping by. You may also occasionally set a study to non-discoverable temporarily so you can try it out as a participant without actually recruiting!

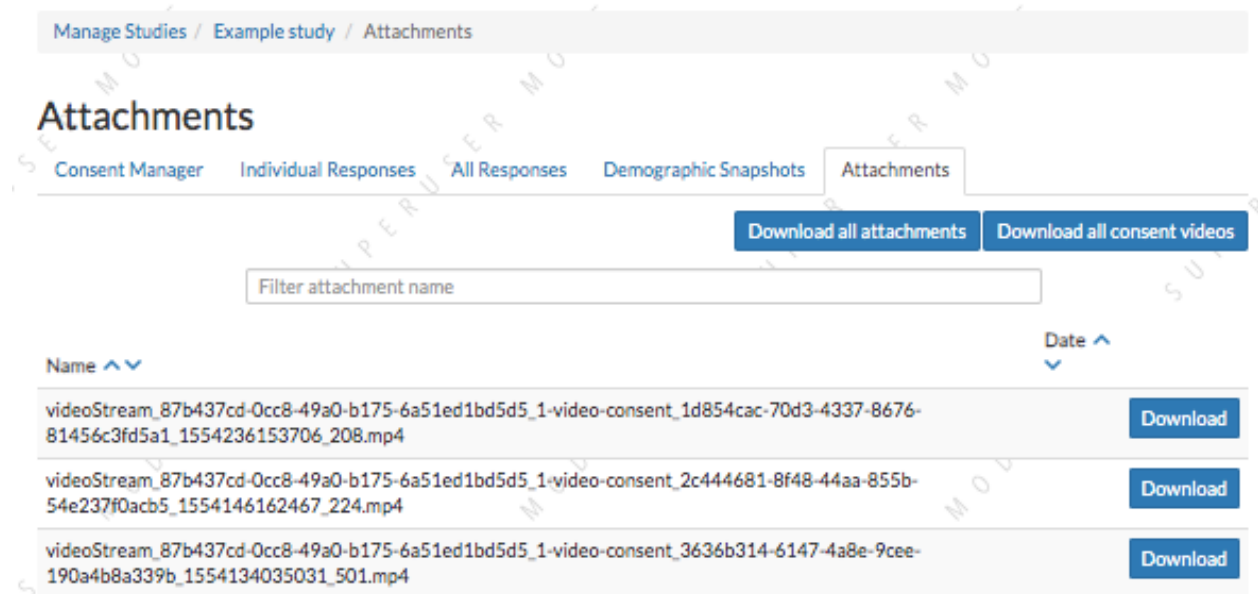


### 1.4.13 Build study

This needs to be a valid JSON block describing the different frames (pages) of your study, and the sequence. You can add these later under `/exp/studies/<study_id>/edit/build/`. For detailed information about specifying your study protocol, see [Building an Experiment](#).

### 1.4.14 Study type

The study type is the application you're using to enable participants to take a study. Right now, we just have one option, the [Ember Frame Player](#). It's an ember app that can talk to our API. All the frames in the experiment are defined in Ember and there is an `exp-player` component that can cycle through these frames. For details, see [Editing study type](#)



## 1.5 Building your experiment

### 1.5.1 Preliminaries: JSON format

Researchers specify the protocol for a Lookit study by providing a JSON (JavaScript Object Notation) object on the Experimenter interface, which is interpreted according to a JSON Schema (<http://json-schema.org/>) designed for Lookit studies. A [JSON schema](#) describes a class of JSON objects, indicating what type of data to expect and require.

If you are unfamiliar with the JSON format, you may want to spend a couple minutes reading the introduction here: <http://www.json.org/>.

No programming is required to design a study: JSON is a simple, human-readable text format for describing data (see <http://www.json.org/>). A JSON object is an unordered set of key – value pairs, with the following rules

- The object itself is enclosed in curly braces.
- Keys are unique strings enclosed in double quotes.
- A key and value are separated by a colon.
- Key-value pairs are separated by commas.

A JSON value can be any of the following: a string (enclosed in double quotes), a number, a JSON object (as described above), an array (an ordered list of JSON values, separated by commas and enclosed by square brackets), true, false, or null. There are no requirements for specific formatting of a JSON document (any whitespace not part of a string is ignored). Here is an example JSON object to illustrate these principles:

```
{
  "name": "Jane",
  "age": 43,
  "favoritefoods": [
    "eggplant",
    "apple",
    "lima beans"
  ],
  "allergies": {
    "peanut": "mild",
    "shellfish": "severe"
  }
}
```

The keys are the strings name, age, favoritefoods, and allergies. Favorite foods are stored as an array, or ordered list; allergies are stored as a JSON object mapping food names to severity of reaction. The same object could also be written as follows, in a different order and with none of the formatting: `{ "age": 43, "allergies": { "peanut": "mild", "shellfish": "severe" }, "name": "Jane", "favoritefoods": ["eggplant", "apple", "lima beans"] }`

A helpful resource to check your JSON Schema for simple errors like missing or extra commas, unmatched braces, etc. is [jsonlint](#).

## 1.5.2 Experiment structure

To define what actually happens in your study, click ‘Edit study’ from your study detail page, and scroll down to the ‘Build study - add JSON’ field:

The screenshot shows a web form for editing a study. At the top, there's a section for 'Researcher Contact Information' with a text input containing 'Jane Smith (contact: jsmith@science.edu)'. Below this is a text block explaining the format: 'This should give the name of the PI for your study, and an email address where the PI or study staff can be reached with questions. Format: PIs Name (contact: youremail@lab.edu)'. There is a checked checkbox labeled 'Discoverable - Do you want this study to be publicly discoverable on Lookit once activated?'. Below that is the 'Build Study - Add JSON' section, which contains a text area with a JSON snippet: `{"frames": {"exit-survey": {"kind": "exp-lookit-exit-survey", "debriefing": {"text": "H`. To the right of the text area is a blue button labeled 'Click to edit'. Below the text area is a line of text: 'Add the frames of your study as well as the sequence of those frames. This can be added later.' At the bottom of the form are three buttons: 'See Preview' (blue), 'Cancel' (white), and 'Save Changes' (red).

Click on this field to bring up the experiment editor view. Here is where you define the structure of your experiment using a JSON document.

Studies on Lookit are broken into a set of fundamental units called **frames**, which can also be thought of as “pages” of the study. A single experimental trial (e.g. looking time measurement) would generally be one frame, as are the video consent procedure and exit survey. Your JSON must have two keys: `frames` and `sequence`. The `frames` value defines the frames used in this study: it must be a JSON object mapping frame nicknames (any unique strings chosen

by the researcher) to frame objects (defined next). The `sequence` value must be an ordered list of the frames to use in this study; values in this list must be frame nicknames from the “frames” value.

Here is the JSON for a very minimal Lookit study:

```
{
  "frames": {
    "my-consent-frame": {
      "kind": "exp-video-consent",
      "prompt": "I agree to participate",
      "blocks": [
        {
          "title": "About the study",
          "text": "This isn't a real study."
        }
      ]
    },
    "my-exit-survey": {
      "kind": "exp-lookit-exit-survey",
      "debriefing": {
        "title": "Thank you!",
        "text": "You participated."
      }
    }
  },
  "sequence": [
    "my-consent-frame",
    "my-exit-survey"
  ]
}
```

This JSON specifies a Lookit study with two frames, consent and an exit survey. Note that the frame nicknames `my-consent-frame` and `my-exit-survey` that are defined in `frames` are also used in the `sequence`. Frames may be specified but not used in `sequence`. Here’s the object associated with the `my-exit-survey` frame:

```
{
  "kind": "exp-lookit-exit-survey",
  "debriefing": {
    "title": "Thank you!",
    "text": "You participated."
  }
}
```

Within each frame object, a `kind` must be specified. This determines the frame type that will be used. Additional data may be included in the frame object to customize the behavior of the frame, for instance to specify instruction text or the stimuli to use for a test trial. The keys that may (or must) be included in a frame object are determined by the frame type; each frame definition includes a JSON Schema describing the expected data to be passed. Multiple frames of the same kind may be included in a study – for instance, test trials using different stimuli.

The separation of frame definitions and sequence allows researchers to easily and flexibly edit and test study protocols – for instance, the order of frames may be altered or a particular frame removed for testing purposes without altering any frame definitions.

### 1.5.3 Developing your study: how to try it out as you go

When you first create your study, you’ll need to click ‘Build preview dependencies’ on the study edit page and wait 5-10 minutes for your own special study environment to be created. This will “freeze” the code used for your study so

that continuing development and changes to the experiment frame code won't affect how your study works. (You can always update if you want to - see [Editing study type](#)). You do not need to build preview dependencies again unless you want to update the study type.

Once you've built preview dependencies once, you can click 'See preview' after saving your study JSON and you will be taken to a preview version of your study so that you can see what it looks like to a participant! As you write the JSON document for your study, you can click 'See preview' again or just refresh the preview window to see how the changes look.

If something isn't working as expected, you can try opening up the Javascript console in your web browser (Chrome: three vertical dots -> More tools -> Developer tools; Firefox: hamburger menu -> Web Developer -> Web Console) to see if there is an error message that makes sense - e.g., a frame type that isn't defined, or an attempt to load an image that doesn't exist.

As you work on a particular frame like a survey, you probably don't want to click through every bit of your study to get to it each time you make a change! You can put the frame of interest at the very start of your study by inserting it at the very start of the 'sequence' you've defined in your JSON. Then when you're satisfied with that frame, just put it back in order.

### 1.5.4 Finding and using specific frames

For the most current documentation of individual frames available to use, please see [the frame documentation](#).

For each frame, you will find an **example** of using it in a JSON schema; documentation of the **properties** which can be defined in the schema; and, under Methods / serializeContent, a description of the **data** this frame records. Any frame-specific **events** that are recorded and may be included in the eventTimings object sent with the data are also described.

### 1.5.5 A Lookit study schema: general principles and instructions

A typical Lookit study might contain the following frame types:

1. **exp-video-config** - This is a standard frame type that almost everyone should just stick at the very start of their study. It requires no customization; we'll maintain troubleshooting directions everyone can share.
2. **exp-lookit-video-consent** - A video consent frame. Your study needs to use this frame and it should come right after video configuration, before getting into the rest of the study. You need to specify some text fields to use this, regarding study-specific procedures, compensation, etc. These will be inserted into the consent document. If you need to show your IRB exactly what your consent document will look like, enter your text snippets, preview your study, and copy the document (or use the download button to get a PDF).
3. **exp-lookit-text** Now we're into optional frames that will vary by study. Most existing studies have started off with a text 'overview' of the study using a frame like this. The shorter this can be, the better - it's the equivalent of "okay, we're ready to get started, we're going to do X, Y, Z!" in the lab. Writing this text, and any instructions, tends to be more time-consuming than researchers expect: in contrast to an in-lab study, you can't easily tune what you say to the individual parent and answer just the questions they bring up. And you don't want to overwhelm them with a wall of text while they try to hold a squirmy baby! **We strongly recommend treating this as a serious writing/design exercise**, and going through a few rounds of 'play-testing' with colleagues/family to make sure everything is as clear and concise as possible.
4. **exp-lookit-preview-explanation** If you are showing children images/videos and you are going to ask the parents **not** to look at those stimuli, we strongly advise that you provide parents an opportunity to preview all of the stimuli that might be shown so they can decide if they're okay with that. This is both a reasonable courtesy (who knows what unusual phobia a child has, or what image you think is totally innocuous but turns out to offend a particular family for an unanticipated reason) and practical for data quality (parents will be less inclined to peek

if they know roughly what’s going on). If you want to show a preview, you’ll use an “explanation” frame like this offering the parent an option to preview stimuli, followed immediately by...

5. [exp-video-preview](#) - the actual video preview frame where you specify a list of videos/images and their captions.
6. [exp-lookit-survey](#) Perhaps you want to collect some information (here or later on) from the parent that isn’t included in the child or demographic data you’ll have automatic access to - how much of which languages they speak in the home, motor milestones, whether their child likes Kermit or Oscar better, etc. You can use a survey frame to do that!
7. [exp-video-config-quality](#) Once you’re almost ready to start your actual ‘test’ procedures, you may want to guide the parent through webcam setup optimization, especially if you need the parent and child in a particular position. We provide some default instructions intended for preferential looking but would recommend making your own images/instructions if you can - ours aren’t great.
8. [exp-lookit-instructions](#) Instead or in addition, you may want a frame like this to give some final instructions to the parent before your ‘test’ procedures start! You can show text, videos, audio, show the user’s webcam, etc. Make sure you have indicated here or earlier that the family is free to leave at any point and how they can do that. (Ctrl-X, F1, or closing the tab/window but then staying on the page will all bring up a “really exit?” dialog - you don’t need to note all methods.)
9. [Study-specific frames, e.g. [exp-lookit-story-page](#), [exp-lookit-preferential-looking](#), [exp-lookit-dialogue-page](#); generally, a sequence of these frames would be put together with a randomizer]
10. [exp-lookit-exit-survey](#) This is a required frame and should be the last thing in your study. This is where participants will select a privacy level for their video and indicate whether data can be shared on Databrary. (If you don’t have IRB/institutional approval to share on Databrary yet, it’s still fine to ask this; worst case you don’t share data you had permission to share. Best case it’ll smooth the process of asking your IRB retroactively if you want to!) Your participants will also have the option to withdraw video beyond the consent video entirely - this is rare (<1 percent of responses). These video settings are provided at the end, rather than the start, of the study so that parents already know roughly what happened and can better judge how comfortable they are with the video being shared. (E.g., “did my child pick his nose the whole time?”)

The ‘debriefing’ field of this frame is **very important!** This is a chance to explain the purpose of your study and how the family helped; at this point it’s more obvious to the participant that skimming the info is fine if they’re not super-interested, so you can elaborate in ways you might have avoided ahead of time in the interest of keeping instructions short. You may want to mention the various conditions kids were assigned to if you didn’t before, and try to head off any concerns parents might have about how their child ‘did’ on the study, especially if there are ‘correct’ answers that will have been obvious to a parent. It’s great if you can link people to a layperson-accessible article on a related topic - e.g., media coverage of one of your previous studies in this research program, a talk on Youtube, a parenting resource.

If you are compensating participants, restate what the compensation is (and any conditions), and let them know when to expect their payment! E.g.: “To thank you for your participation, we’ll be emailing you a \$4 Amazon gift card - this should arrive in your inbox within the next week after we confirm your consent video and check that your child is in the age range for this study. (If you don’t hear from us by then, feel free to reach out!) If you participate again with another child in the age range, you’ll receive one gift card per child.”

## 1.5.6 Randomizer frames

Generally, you’ll want to show slightly different versions of the study to different participants: perhaps you have a few different conditions, and/or need to counterbalance the order of trials or left/right position of stimuli. To do this, you’ll use a special frame called a **randomizer** to select an appropriate sequence of frames for a particular trial. A randomizer frame is automatically expanded to a list of frames, so that for instance you can specify your 12 looking-time trials all at once.

See [here](#) for complete documentation of available randomizers.

To use a randomizer frame, set the frame "kind" to "choice" and "sampler" to the appropriate type of randomizer. We will focus here on the most commonly-used and general randomizer type, called `random-parameter-set`.

To select this randomizer, you need to define a frame that has the appropriate "kind" and "sampler":

```
{
  ...
  "frames": {
    ...
    "test-trials": {
      "sampler": "random-parameter-set",
      "kind": "choice",
      ...
    }
  }
}
```

In addition, there are three special properties you need to define to use `random-parameter-set`: `frameList`, `commonFrameProperties`, and `parameterSets`.

“**frameList**” is just what it sounds like: a list of all the frames that should be generated by this randomizer. Each frame is a JSON object just like you would use in the overall schema, with two differences:

- You can define default properties, to share across all of the frames generated by this randomizer, in the JSON object `commonFrameProperties` instead, as a convenience.

You can use placeholder strings for any of the properties in the frame; they will be replaced based on the values in the selected `parameterSet`.

“**parameterSets**” is a list of mappings from placeholder strings to actual values. When a participant starts your study, one of these sets will be randomly selected, and any parameter values in the `frameList` (including `commonFrameProperties`) that match any of the keys in this parameter set will be replaced.

Let’s walk through an example of using this randomizer. Suppose we start with the following JSON document describing a study that includes instructions, an experimental manipulation asking participants to think about how delicious broccoli is, and an exit survey:

```
{
  "frames": {
    "instructions": {
      "id": "text-1",
      "blocks": [
        {
          "text": "Some introductory text about this study."
        },
        {
          "text": "Here's what's going to happen! You're going to think_
↪about how tasty broccoli is."
        }
      ],
      "showPreviousButton": false,
      "kind": "exp-lookit-text"
    },
    "manipulation": {
      "id": "text-2",
      "blocks": [
        {
          "text": "Think about how delicious broccoli is."
        }
      ],

```

(continues on next page)

(continued from previous page)

```

        {
            "text": "It is so tasty!"
        }
    ],
    "showPreviousButton": true,
    "kind": "exp-lookit-text"
},
"exit-survey": {
    "debriefing": {
        "text": "Thank you for participating in this study! ",
        "title": "Thank you!"
    },
    "id": "exit-survey",
    "kind": "exp-lookit-exit-survey"
}
},
"sequence": [
    "instructions",
    "manipulation",
    "exit-survey"
]
}

```

But what we really want to do is have some kids think about how tasty broccoli is, and others think about how yucky it is! We can use a `random-parameter-set` frame to replace both text frames:

```

{
    "frames": {
        "instruct-and-manip": {
            "sampler": "random-parameter-set",
            "kind": "choice",
            "id": "instruct-and-manip",
            "frameList": [
                {
                    "blocks": [
                        {
                            "text": "Some introductory text about this study."
                        },
                        {
                            "text": "INTROTEXT"
                        }
                    ],
                    "showPreviousButton": false
                },
                {
                    "blocks": [
                        {
                            "text": "MANIP-TEXT-1"
                        },
                        {
                            "text": "MANIP-TEXT-2"
                        }
                    ],
                    "showPreviousButton": true
                }
            ]
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        "commonFrameProperties": {
            "kind": "exp-lookit-text"
        },
        "parameterSets": [
            {
                "INTROTEXT": "Here's what's going to happen! You're going to_
↪think about how tasty broccoli is.",
                "MANIP-TEXT-1": "Think about how delicious broccoli is.",
                "MANIP-TEXT-2": "It is so tasty!"
            },
            {
                "INTROTEXT": "Here's what's going to happen! You're going to_
↪think about how disgusting broccoli is.",
                "MANIP-TEXT-1": "Think about how disgusting broccoli is.",
                "MANIP-TEXT-2": "It is so yucky!"
            }
        ]
    },
    "exit-survey": {
        "debriefing": {
            "text": "Thank you for participating in this study! ",
            "title": "Thank you!"
        },
        "id": "exit-survey",
        "kind": "exp-lookit-exit-survey"
    }
},
"sequence": [
    "instruct-and-manip",
    "exit-survey"
]
}

```

Notice that since both of the frames in the `frameList` were of the same kind, we could define the kind in `commonFrameProperties`. We no longer define `id` values for the frames, as they will be automatically identified as `instruct-and-manip-1` and `instruct-and-manip-2`.

When the “instruct-and-manip” randomizer is evaluated, the Lookit experiment player will start with the `frameList` and add the key-value pairs in `commonFrameProperties` to each frame (not overwriting existing pairs):

```

[
  {
    "kind": "exp-lookit-text",
    "blocks": [
      {
        "text": "Some introductory text about this study."
      },
      {
        "text": "INTROTEXT"
      }
    ],
    "showPreviousButton": false
  },
  {
    "kind": "exp-lookit-text",
    "blocks": [
      {

```

(continues on next page)



(continued from previous page)

```

        "text": "MANIP-TEXT-1"
      },
      {
        "text": "MANIP-TEXT-2"
      }
    ],
    "showPreviousButton": true
  }
]

```

Next, one of the two objects in `parameterSets` is selected randomly. (By default, parameter sets are weighted equally, but `parameterSetWeights` can be provided as an optional key in the `random-parameter-set` frame. If provided, `parameterSetWeights` should be an array of relative weights for the parameter sets, corresponding to the order they are listed. For instance, if we wanted 75% of participants to think about how tasty broccoli is, we could set `parameterSetWeights` to `[3, 1]`. This allows uneven condition assignment where needed to optimize power, as well as allowing researchers to stop testing conditions that already have enough participants as data collection proceeds.)

Suppose that in this case the second parameter set is selected:

```

{
  "INTROTEXT": "Here's what's going to happen! You're going to think about how
↳disgusting broccoli is.",
  "MANIP-TEXT-1": "Think about how disgusting broccoli is.",
  "MANIP-TEXT-2": "It is so yucky!"
}

```

Now we return to the list of frames, and wherever any value matches one of the keys in the `parameterSet` (even if that value is nested in another object), it is replaced by the corresponding value from the `parameterSet`, yielding the following final list of frames:

```

[
  {
    "kind": "exp-lookit-text",
    "blocks": [
      {
        "text": "Some introductory text about this study."
      },
      {
        "text": "Here's what's going to happen! You're going to think about
↳how disgusting broccoli is."
      }
    ],
    "showPreviousButton": false
  },
  {
    "kind": "exp-lookit-text",
    "blocks": [
      {
        "text": "Think about how disgusting broccoli is."
      },
      {
        "text": "It is so yucky!"
      }
    ],
    "showPreviousButton": true
  }
]

```

(continues on next page)

(continued from previous page)

```

    }
  ]
}

```

### 1.5.6.1 Nested randomizers

In more complex experimental designs, the frames created by a randomizer may themselves be randomizers! This nesting allows more modular specification: for instance, a study might have ten test trials, each of which consists of three phases. The “outer” randomizer could then generate a `frameList` of ten randomizer frames, each of which would be resolved in turn into three frames. Below is a simplified example with only two test trials, each of which has three phases:

Here’s an example. Notice that `"kind": "choice"`, `"sampler": "random-parameter-set"`, `"frameList": ...`, and `commonFrameProperties` are `commonFrameProperties` of the outer frame `nested-trials`. That means that every “frame” we’ll create as part of `nested-trials` will itself be a `random-parameter-set` generated list with the same frame sequence, although we’ll be substituting in different parameter values. (This doesn’t have to be the case - we could show different types of frames in the list - but in the simplest case where you’re using `randomParameterSet` just to group similar repeated frame sequences, this is probably what you’d do.) The only thing that differs across the two (outer-level) **trials** is the `parameterSet` used, and we list only one parameter set for each trial, to describe (deterministically) how the outer-level `parameterSet` values should be applied to each particular frame.

```

{
  "sampler": "random-parameter-set",
  "frameList": [
    {
      "parameterSets": [
        {
          "NTRIAL": 1,
          "PHASE1STIM": "T1P1",
          "PHASE2STIM": "T1P2",
          "PHASE3STIM": "T1P3"
        }
      ]
    },
    {
      "parameterSets": [
        {
          "NTRIAL": 2,
          "PHASE1STIM": "T2P1",
          "PHASE2STIM": "T2P2",
          "PHASE3STIM": "T2P3"
        }
      ]
    }
  ],
  "parameterSets": [
    {
      "T1P1": "mouse",
      "T1P2": "rat",
      "T1P3": "chipmunk",
      "T2P1": "horse",
      "T2P2": "goat",
      "T2P3": "cow"
    }
  ],
}

```

(continues on next page)

(continued from previous page)

```

    {
      "T1P1": "guppy",
      "T1P2": "tadpole",
      "T1P3": "goldfish",
      "T2P1": "whale",
      "T2P2": "manatee",
      "T2P3": "shark"
    }
  ],
  "commonFrameProperties": {
    "sampler": "random-parameter-set",
    "frameList": [
      {
        "nPhase": 1,
        "animal": "PHASE1STIM"
      },
      {
        "nPhase": 2,
        "animal": "PHASE2STIM"
      },
      {
        "nPhase": 3,
        "animal": "PHASE3STIM"
      }
    ],
    "commonFrameProperties": {
      "nTrial": "NTRIAL",
      "kind": "question-about-animals-frame"
    }
  }
}

```

To evaluate this experiment frame, the Lookit experiment player starts with the list of frames in the outer `frameList`, adding the key:value pairs in the outer `commonFrameProperties` to each frame, which yields the following list of frames:

```

[
  {
    "parameterSets": [
      {
        "NTRIAL": 1,
        "PHASE1STIM": "T1P1",
        "PHASE2STIM": "T1P2",
        "PHASE3STIM": "T1P3"
      }
    ],
    "sampler": "random-parameter-set",
    "frameList": [
      {
        "nPhase": 1,
        "animal": "PHASE1STIM"
      },
      {
        "nPhase": 2,
        "animal": "PHASE2STIM"
      }
    ]
  }
]

```

(continues on next page)

(continued from previous page)

```

    },
    {
      "nPhase": 3,
      "animal": "PHASE3STIM"
    }
  ],
  "commonFrameProperties": {
    "nTrial": "NTRIAL",
    "kind": "question-about-animals-frame"
  }
},
{
  "parameterSets": [
    {
      "NTRIAL": 2,
      "PHASE1STIM": "T2P1",
      "PHASE2STIM": "T2P2",
      "PHASE3STIM": "T2P3"
    }
  ],
  "sampler": "random-parameter-set",
  "frameList": [
    {
      "nPhase": 1,
      "animal": "PHASE1STIM"
    },
    {
      "nPhase": 2,
      "animal": "PHASE2STIM"
    },
    {
      "nPhase": 3,
      "animal": "PHASE3STIM"
    }
  ],
  "commonFrameProperties": {
    "nTrial": "NTRIAL",
    "kind": "question-about-animals-frame"
  }
}
]

```

One of the two (outer) `parameterSets` is then selected randomly; suppose the second one (aquatic instead of land animals) is selected. Now any substitutions are made based on the keys in this `parameterSet`. The first frame in the sequence is now:

```

{
  "parameterSets": [
    {
      "NTRIAL": 1,
      "PHASE1STIM": "guppy",
      "PHASE2STIM": "tadpole",
      "PHASE3STIM": "goldfish"
    }
  ],
  "sampler": "random-parameter-set",

```

(continues on next page)

(continued from previous page)

```

"frameList": [
  {
    "nPhase": 1,
    "animal": "PHASE1STIM"
  },
  {
    "nPhase": 2,
    "animal": "PHASE2STIM"
  },
  {
    "nPhase": 3,
    "animal": "PHASE3STIM"
  }
],
"commonFrameProperties": {
  "nTrial": "NTRIAL",
  "kind": "question-about-animals-frame"
}
}

```

Next, each frame is expanded since it is in turn another randomizer (due to "sampler": "random-parameter-set"). The frame above, representing Trial 1, will be turned into three frames. First, again, we start with the frameList, and merge the commonFrameProperties into each frame:

```

[
  {
    "nPhase": 1,
    "animal": "PHASE1STIM",
    "nTrial": "NTRIAL",
    "kind": "question-about-animals-frame"
  },
  {
    "nPhase": 2,
    "animal": "PHASE2STIM",
    "nTrial": "NTRIAL",
    "kind": "question-about-animals-frame"
  },
  {
    "nPhase": 3,
    "animal": "PHASE3STIM",
    "nTrial": "NTRIAL",
    "kind": "question-about-animals-frame"
  }
]

```

Finally, a parameter set is selected from parameterSets. Only one parameter set is defined for this trial, which is deliberate; it simply selects the correct stimuli for this trial. Substituting in the values from the parameter set yields the following list of frames:

```

[
  {
    "nPhase": 1,
    "animal": "guppy",
    "nTrial": 1,
    "kind": "question-about-animals-frame"
  },

```

(continues on next page)

(continued from previous page)

```

{
  "nPhase": 2,
  "animal": "tadpole",
  "nTrial": 1,
  "kind": "question-about-animals-frame"
},
{
  "nPhase": 3,
  "animal": "goldfish",
  "nTrial": 1,
  "kind": "question-about-animals-frame"
}
]

```

The random-parameter-set randomizer is expected to be general enough to capture most experimental designs that researchers put on Lookit, but additional more specific randomizers will also be designed to provide simpler syntax for common use cases.

### 1.5.7 Conditional logic

In some cases, what happens next in your study will need to depend on what has happened so far, what happened during previous sessions of the study, and/or information about the participant. For instance, perhaps you want to move on from a training segment after the participant answers three questions in a row correctly, or you want to start with an eligibility survey and only route people to the rest of the study if they meet detailed criteria. Or maybe you just want to personalize instructions or stimuli with the child's name and gender! All Lookit frames allow you to provide either or both of the following properties to flexibly specify conditional behavior:

1. **generateProperties**: Provide a function that takes `expData`, `sequence`, `child`, `pastSessions`, and `conditions` objects, and returns an object representing any additional properties that should be used by this frame - e.g., the frame type, text blocks, whether to do recording, etc. (In principle a `generateProperties` function could conditionally assign `selectNextFrame`, although we do not know of a use case where this is necessary.)
2. **selectNextFrame**: Provide a function that takes `frames`, `frameIndex`, `expData`, `sequence`, `child`, and `pastSessions` and returns that frame index to go to when using the 'next' action on this frame. For instance, this allows you to skip to the end of the study (or a frame of a particular type) if the child has gotten several questions correct.

Each of these properties is specified as a string, which must define a Javascript function of the specified arguments. `generateProperties` is called when the frame is initialized, and `selectNextFrame` is called upon proceeding to the next frame.

Formal documentation for these properties is linked above. However, in practice, if you want to add some conditional behavior and are wondering e.g. how to get the child's first name or birthday, or how to determine what condition the child is in, it may be easiest to get started by adding a dummy function like the following to the frame in question:

```

"generateProperties": "function(expData, sequence, child, pastSessions, conditions)
↳{console.log(expData); console.log(sequence); console.log(child); console.
↳log(pastSessions); console.log(conditions); return {};}"

"selectNextFrame": "function(frames, frameIndex, frameData, expData, sequence, child,
↳pastSessions) {console.log(frames); console.log(frameIndex); console.log(frameData);
↳ console.log(expData); console.log(sequence); console.log(child); console.
↳log(pastSessions); return (frameIndex + 1);} "

```

These functions just log each of the arguments they're given the Javascript console; there you can take a look and play around with how you'd access and manipulate the properties you need. The `generateProperties` function above just return an empty object, not assigning any properties. The `selectNextFrame` function just returns `frameIndex + 1`, i.e. says the next frame should be the one after this one, not changing the frame's regular behavior.

Although you'll need to enter these properties as single-line strings in the Lookit study editor, they are obviously not very readable that way! You can go from a single-line string back to something readable using a Javascript 'beautifier' like [this](#) - you may want to do that to better understand the examples below. When you are writing your own functions, you can write them on multiple lines in your text editor and then either strip out the line breaks using your text editor or one of many online tools like [this](#).

### 1.5.7.1 Example: eligibility survey

Here is an example of a situation where you might want to determine the sequence of frames in a study and/or behavior of those frames based on data collected earlier in the study. Suppose you want to start off with a survey to determine eligibility, using criteria that go beyond what is available in Lookit child/demographic surveys and usable for automatic eligibility detection. (Perhaps your study is very involved or won't make sense to people who don't meet criteria, so you don't want to just have everyone participate and filter the data afterwards.)

A similar approach would be appropriate if you wanted to customize the behavior of the study based on user input - e.g., using the child's favorite color for stimuli, let the family choose which game they want to play this time, or let the family choose whether to 'actually' participate (and have video recorded) or just see a demo.

This example has three top-level frames: an eligibility survey, a study procedure (which depends on eligibility as determined from the survey), and an exit survey (with debriefing text that depends on eligibility too).

```
{
  "frames": {
    "exit-survey": {
      "kind": "exp-lookit-exit-survey",
      "generateProperties": "function(expData, sequence, child, pastSessions)
↪{var eligible = expData['1-study-procedure']['generatedProperties']['ELIGIBLE']; if
↪(eligible) { return { 'debriefing': { 'text': 'In this study, we
↪were looking at why babies love cats. Your child actually participated. A real
↪debriefing would be more detailed.', 'title': 'Thank you!' } }; } else { return {
↪'debriefing': { 'text': 'In this study, we would have looked at why
↪your child loved cats. Your child did not actually participate though. A real
↪debriefing would make more sense.', 'title': 'Thank you!' } }; }}"
    },
    "eligibility-survey": {
      "kind": "exp-lookit-survey",
      "formSchema": {
        "schema": {
          "type": "object",
          "title": "Eligibility survey",
          "properties": {
            "nCats": {
              "type": "integer",
              "title": "How many cats do you have?",
              "maximum": 200,
              "minimum": 0,
              "required": true
            },
            "loveCats": {
              "enum": [
                "yes",
```

(continues on next page)

(continued from previous page)

```

        "no"
      ],
      "type": "string",
      "title": "Does your baby love cats?",
      "required": true
    }
  },
  "options": {
    "fields": {
      "nCats": {
        "numericEntry": true
      },
      "loveCats": {
        "type": "radio",
        "message": "Please answer this question.",
        "validator": "required-field"
      }
    }
  },
  "nextButtonText": "Continue"
},
"study-procedure": {
  "kind": "exp-frame-select",
  "frameOptions": [
    {
      "kind": "exp-frame-select",
      "frameOptions": [
        {
          "kind": "exp-lookit-text",
          "blocks": [
            {
              "emph": true,
              "text": "Let's start the study!"
            },
            {
              "text": "Some info about cats..."
            }
          ]
        },
        {
          "kind": "exp-lookit-text",
          "blocks": [
            {
              "emph": true,
              "text": "Cats are great"
            },
            {
              "text": "We are measuring how much your child_
↪ loves cats now. Beep boop!"
            }
          ]
        }
      ]
    }
  ]
},
{

```

(continues on next page)



(continued from previous page)

```

        "kind": "exp-lookit-text",
        "blocks": [{
            "emph": true,
            "text": "Your child is not eligible for this study"
        },
        {
            "text": "Either you do not have any cats or your child_
↪does not love cats."
        }
    ],
    "generateProperties": "function(expData, sequence, child, pastSessions)
↪{var formData = expData['0-eligibility-survey'].formData; if (formData.nCats >= 1 &&
↪formData.loveCats == 'yes') { console.log('eligible'); return { 'whichFrames': 0,
↪'ELIGIBLE': true } } else { console.log('ineligible'); return { 'whichFrames': 1,
↪'ELIGIBLE': false } } }"
    },
    "sequence": [
        "eligibility-survey",
        "study-procedure",
        "exit-survey"
    ]
}

```

Here's how it works:

1. The study procedure is set up as an `exp-frame-select` frame, and we decide on-the-spot which of the two `frameOptions` to use based on the data in the survey by providing a `generateProperties` function that returns a value for `whichFrames`. The function `generateProperties` is called when we get to the `study-procedure` frame, and the key-value pairs it returns get added to the other parameters for this frame (like `kind` and `frameOptions`). In this case, it checks to see whether the survey says the family has at least one cat *and* the child loves cats; in that case, the child is eligible to participate.

Additionally, the object `generateProperties` returns is stored under the key `generatedProperties` in `expData` for this frame, so that we can use the output later. That's why we also include either `'ELIGIBLE': true` or `'ELIGIBLE': false` - that way we can reuse this determination later on in another `generateProperties` function.

2. If the child isn't eligible, the `study-procedure` frame just resolves to a single `exp-lookit-text` frame, at index 1 of `frameOptions`. If the child is eligible, the `study-procedure` frame resolves to a second `exp-frame-select` frame, which just serves to bundle up a few text frames. We don't provide `whichFrames`, so all of the `frameOptions` listed will be shown in order. (We could also have set this up without a nested `exp-frame-select` frame, e.g. by putting all three `exp-lookit-text` frames in the outer `frameOptions` and saying that if the child is eligible, use `whichFrames = [0, 1]`, and if not, `whichFrames = 2`.)
3. After the study procedure is done, everyone goes to an exit survey. The `generateProperties` function of the exit survey returns different debriefing text based on the stored `ELIGIBLE` value we defined earlier.

Note that the data stored in `expData`` will include frame data for the `exp-frame-select` frames, even though these are not actually displayed as frames separate from the contents they resolve to. For a child who is eligible, the keys in `expData` will be:

- 0-eligibility-survey
- 1-study-procedure (the outer `exp-frame-select` frame)

- 1-study-procedure-0 (the inner exp-frame-select frame)
- 1-study-procedure-0-0 (the first exp-lookit-text frame)
- 1-study-procedure-0-1 (the second exp-lookit-text frame)

### 1.5.7.2 Example: waiting for successful training

Sometimes, you might want to skip ahead to the next section of an experiment once certain criteria are met. For instance:

- you might have a study where questions get harder and harder over time, and you just want to keep asking until the child gets N wrong in a row
- you might want to have a “training” section that allows the family to practice until they’re ready
- you might want to make one section of a study optional, and skip over it if the parent opts to (or if it’s not applicable to them)

Here’s an example study where we wait for the child to get two “training” questions right, then proceed to a “test” question:

```
{
  "frames": {
    "exit-survey": {
      "kind": "exp-lookit-exit-survey",
      "debriefing": {
        "title": "Thank you!",
        "text": "Thank you for participating in this study"
      }
    },
    "training-question-block": {
      "kind": "exp-frame-select",
      "frameOptions": [
        {}, {}, {}, {}, {}, {}, {}, {}, {}, {}
      ],
      "commonFrameProperties": {
        "kind": "exp-lookit-survey",
        "generateProperties": " function(expData, sequence, child,
→pastSessions) {
→                                var n = Math.floor(Math.random() * Math.floor(20));
→                                var m = Math.floor(Math.random() * Math.floor(20));
→return {
→                                'formSchema': {
→                                'schema': {
→                                'type': 'object',
→                                'title': 'Math
→practice question',
→                                'properties': {
→                                'enum': [
→                                'low',
→                                'high'
→                                ],
→                                'title
→': 'What is ' + n + ' plus ' + m + '?',
→                                'required
→': true
→                                }
→                                },
→                                'options': {
→                                'fields
→': {
→                                'add': {
→                                'type': 'radio',
→                                'optionLabels': [n + m - 1, n + m, n + m + 1],
→                                'message': 'Please answer this question.',
→                                'validator': 'required-field'}}}}}}",
        "selectNextFrame": "function(frames, frameIndex, frameData, expData,
→sequence, child, pastSessions) {
→                                var testFrame = 0; for (var iFrame = 0; iFrame
→< frames.length; iFrame++) {if (frames[iFrame]['id'].indexOf('test-question') != -
→1) {testFrame = iFrame; break;}} if ((sequence.length >= 3) &&
→(expData[sequence[sequence.length - 2]]['formData']['add'] == 'correct' ) &&
→(expData[sequence[sequence.length - 1]]['formData']['add'] == 'correct' (continues on next page)
→return testFrame;
→                                }
→                                else {
→                                return frameIndex + 1;
→                                }}"
      }
    }
  }
}
```

(continued from previous page)

```

    }
  },
  "test-question": {
    "kind": "exp-lookit-survey",
    "generateProperties": " function(expData, sequence, child, pastSessions)
↪{
    var n = Math.floor(Math.random() * Math.floor(20));
↪var m = Math.floor(Math.random() * Math.floor(20));
↪    return {
↪      'formSchema': {
↪        'type': 'object',
↪        'properties': {
↪          'enum': [
↪            'low',
↪            'high'
↪          ],
↪          'title': 'What is ' +
↪          n + ' minus ' + m + '?',
↪          'required': true
↪        },
↪        'options': {
↪          'subtract': {
↪            'optionLabels': [n - m - 1, n - m, n - m + 1],
↪            'message': 'Please answer this question.',
↪            'validator': 'required-field'}}}}}}"
    },
    "sequence": [
      "training-question-block",
      "test-question",
      "exit-survey"
    ]
  }
}

```

There are three sections in the study: a block of up to 10 training questions, a single test question, and an exit survey. We use an `exp-frame-select` frame to quickly create ten identical training question frames, by putting all of the frame properties into `commonFrameProperties`. We use `generateProperties` not to do anything contingent on the child or study data, but just to programmatically generate the questions - this way we can choose random numbers for each question. Finally, we add a `selectNextFrame` function to the training questions. Let's take a closer look at that function:

```

function(frames, frameIndex, frameData, expData, sequence, child, pastSessions) {
  // First, find the index of the test frame in case we need to go there
  var testFrame = 0;
  for (var iFrame = 0; iFrame < frames.length; iFrame++) {
    if (frames[iFrame]['id'].indexOf('test-question') !== -1) {
      testFrame = iFrame;
      break;
    }
  }
  // If the last two questions were answered correctly, go to test
  if ((sequence.length >= 3) && (expData[sequence[sequence.length - 2]]['formData']['add']
↪== 'correct') && (expData[sequence[sequence.length - 1]]['formData']['add']
↪== 'correct')) {
    return testFrame;
  } else {
    // Otherwise, just go to the next frame
    return frameIndex + 1;
  }
}

```

We first use the list of `frames` to identify the index of the test question. (In this case we could safely assume it's the

second-to-last frame, too. But in a more complex experiment, we might want to find it like this.)

Then we check whether (a) there are already at least 3 frames including this one in the sequence (two practice questions plus the initial `exp-frame-select` frame) and (b) the last two questions including this one were answered correctly. If so, we skip right to the test question!

### 1.5.7.3 Example: personalized story

One of the objects you have access to in your `generateProperties` function is the `child`. This allows you to use child data in selecting stimuli, instructions, or procedures. A simple use case would be personalizing a story (or instructions) using the child’s name and gender. Here’s an example:

```
{
  "frames": {
    "personalized-story": {
      "kind": "exp-lookit-text",
      "generateProperties": "function(expData, sequence, child, pastSessions,
↪conditions) {var childName = child.get('givenName'); var genderedChild; if (child.
↪get('gender') == 'f') {   genderedChild = 'girl';} else if (child.get('gender') ==
↪'m') {   genderedChild = 'boy';} else {genderedChild = 'kiddo';} var line1 = 'Once
↪upon a time, there was a little ' + genderedChild + ' named ' + childName + '.';
↪var line2 = childName + ' loved to draw.'; return {'blocks': [{'text': line1}, {
↪'text': line2}]};"
    },
    "sequence": [
      "personalized-story"
    ]
  }
}
```

### 1.5.7.4 Example: debriefing text that depends on experimental condition

One fairly common and straightforward use case for customizing frames based on data from the experiment is that you might like to debrief parents at the end of the study based on the experimental condition their child was in, just like you would in the lab.

Here’s an example where we have an experimental “procedure” that depends on condition assignment in a `random-parameter-set` frame, and mention the condition in the debriefing text:

```
{
  "frames": {
    "exit-survey": {
      "kind": "exp-lookit-exit-survey",
      "debriefing": {
        "title": "Thank you!",
        "text": "Thank you for participating in this study. Your child was in
↪the "
      },
      "generateProperties": "function(expData, sequence, child, pastSessions,
↪conditions) {if (conditions['1-study-procedure']['conditionNum'] == 0) {return {
↪'debriefing': {'title': 'Thank you!', 'text': 'Your child was in the cats condition.
↪'}};} else {return {'debriefing': {'title': 'Thank you!', 'text': 'Your child was
↪in the dogs condition.'}};} }"
    },
    "study-procedure": {
```

(continues on next page)

(continued from previous page)

```

    "sampler": "random-parameter-set",
    "kind": "choice",
    "frameList": [
      {
        "kind": "exp-lookit-text",
        "blocks": [
          {
            "text": "PROCEDURE_TEXT",
            "title": "PROCEDURE_TITLE"
          }
        ]
      }
    ],
    "parameterSets": [
      {
        "PROCEDURE_TEXT": "All about cats",
        "PROCEDURE_TITLE": "Cats say meow!"
      },
      {
        "PROCEDURE_TEXT": "All about dogs",
        "PROCEDURE_TITLE": "Dogs say woof!"
      }
    ]
  },
  "sequence": [
    "study-procedure",
    "exit-survey"
  ]
}

```

Your debriefing information could also take into account other factors - for instance, if you were conducting a give-N task, you could actually give an automatic estimate of the child’s knower-level or show a chart of their responses! As an exercise, try personalizing the debriefing text to use the child’s name.

## 1.6 Preparing your stimuli

### 1.6.1 Audio and video files

Most experiments will involve using audio and/or video files! You are responsible for creating these and hosting them somewhere (contact MIT if you need help finding a place to put them).

You may find that there are more ‘stimuli’ to create than you’d have in the lab, because in the lab you have the luxury of being present to explain what’s going on. When you design an online study, you may need to record many of the things you’d say in the lab, or create pictures or demos to show how things work. This includes not just storybook audio, but little clips like “We’re almost done!” or “Okay, go ahead and turn around now.”

For basic editing of audio files, if you don’t already have a system in place, we highly recommend [Audacity](#). You can create many “tracks” or select portions of a longer recording using labels, and export them all at once; you can easily adjust volume so it’s similar across your stimuli; and the simple “noise reduction” filter works well. At a minimum, even if these are not ‘stimuli’ per se (e.g., verbal instructions), we recommend

1. Using **noise reduction** to make speech clearer and remove any background ‘buzz’ during pauses. First select a segment of silence to analyze, then apply noise reduction across the whole audio recording; you may need to

play around with the defaults to get excellent noise reduction without distortion, but it does a pretty good job out of the box.

2. Using the **‘amplify’** filter to make all stimuli and instructions approximately equally loud (by default, it makes a segment of audio as loud as possible without clipping).
3. **Trimming** ALL of the silence from the beginning and end of the audio clip. This silence may not be especially noticeable when you simply play the file, but it translates into an unnecessary delay between whenever you trigger the audio file to play in your study and when the relevant sound actually starts.

For editing of video files, we recommend getting comfortable with the command-line tool [ffmpeg](#). It’s a bit of a pain to get used to, but then you’ll be able to do almost anything you can imagine with audio and video files.

## 1.6.2 File formats

To have your media play properly across various web browsers, you will generally need to provide multiple file formats. For a comprehensive overview of this topic, see [MDN](#).

MIT’s standard practice is to provide mp3 and ogg formats for audio, and webm and mp4 (H.264 video codec + AAC audio codec) for video, to cover modern browsers. The easiest way to create the appropriate files, especially if you have a lot to convert, is to use [ffmpeg](#).

Here’s an example command to convert a video file INPUTPATH to mp4 with reasonable quality/filesize and using H.264 & AAC codecs:

```
ffmpeg -i INPUTPATH -c:v libx264 -preset slow -b:v 1000k -maxrate 1000k
-bufsize 2000k -c:a libfdk_aac -b:a 128k
```

And to make a webm file:

```
ffmpeg -i INPUTPATH -c:v libvpx -b:v 1000k -maxrate 1000k -bufsize 2000k -c:a
libvorbis -b:a 128k -speed 2
```

Converting all your audio and video files can be easily automated in Python. Here’s an example script that uses [ffmpeg](#) to convert all the m4a and wav files in a directory to mp3 and ogg files:

```
import os
import subprocess as sp
import sys

audioPath = '/Users/kms/Dropbox (MIT)/round 2/ingroupobligations/lookit stimuli/audio_
↳clips/'

audioFiles = os.listdir(audioPath)

for audio in audioFiles:
    (shortname, ext) = os.path.splitext(audio)
    print shortname
    if not(os.path.isdir(os.path.join(audioPath, audio))) and ext in ['.m4a', '.wav']:
        sp.call(['ffmpeg', '-i', os.path.join(audioPath, audio), \
            os.path.join(audioPath, 'mp3', shortname + '.mp3')])
        sp.call(['ffmpeg', '-i', os.path.join(audioPath, audio), \
            os.path.join(audioPath, 'ogg', shortname + '.ogg')])
```

## 1.6.3 Making dummy stimuli

Sometimes you may not have your stimuli actually ready yet, but you want to make sure your experiment will work as intended once they’re ready. Here’s an example of using [ffmpeg](#) to make some “dummy” images of text to represent

distinct exemplars of various categories. You could also create videos by setting the duration in seconds (here `d=0.01`) to something longer and using an `mp4` or `webm` extension for output instead of `jpg`.

```
import os
import subprocess as sp
import sys

baseDir = '/Users/kms/Desktop/labelsconcepts/img/'

for catDir in ['nov1', 'nov2', 'nov3', 'cats', 'dogs', 'iguanas', 'manatees',
↳ 'squirrels']:
    os.mkdir(os.path.join(baseDir, catDir));
    for iIm in range(1, 12):
        text = catDir + '.' + str(iIm)
        output = os.path.join(baseDir, catDir, str(iIm) + '.jpg')
        sp.call(['ffmpeg', '-f', 'lavfi', '-i', 'color=c=gray:s=640x480:d=0.01', '-vf
↳ ',
                    "drawtext=fontfile=drawtext='fontfile=/Library/Fonts/Arial Black.ttf
↳ ':text='" + text + "':fontsize=64;fontcolor=black:x=10:y=10",
                    output])
```

## 1.6.4 Directory structure

For convenience, many Lookit experiment frames use an `expand-assets` `mixin` that allows you to define a base directory (`baseDir`) as part of the frame definition, so that instead of providing full paths to your stimuli (including multiple file formats) you can give relative paths and specify the audio and/or video formats to expect (`audioTypes` and `videoTypes`).

For instance, the `exp-lookit-story-page` frame allows this - you can see at the very top of the docs that it uses `ExpandAssets`, and under ‘Properties’ you can see the `baseDir`, `audioTypes`, and `videoTypes` arguments.

**Images:** Anything without `://` in the string will be assumed to be a relative image source.

**Audio/video sources:** If you want to provide full paths to stimuli, you will be providing a list of sources, like this:

```
[
  {
    "src": "http://stimuli.org/myAudioFile.mp3",
    "type": "audio/mp3"
  },
  {
    "src": "http://stimuli.org/myAudioFile.ogg",
    "type": "audio/ogg"
  }
]
```

Instead of listing multiple sources, which are generally the same file in different formats, you can alternately list a single string like `"myAudioFile"`.

If you use this option, your stimuli will be expected to be organized into directories based on type.

- **baseDir/img/:** all images (any file format; include the file format when specifying the image path)
- **baseDir/ext/:** all audio/video media files with extension `ext`

**Example:** Suppose you set `"baseDir": "http://stimuli.org/mystudy/"` and then specified an image source as `"train.jpg"`. That image location would be expanded to `http://stimuli.org/mystudy/img/train.jpg`. If you specified that the audio types you were using were `mp3` and `ogg` (the default) by setting `"audioTypes": ["mp3", "ogg"]`, and specified an audio source as `"honk"`, then audio files would be

expected to be located at `http://stimuli.org/mystudy/mp3/honk.mp3` and `http://stimuli.org/mystudy/ogg/honk.ogg`.

## 1.7 Experiment data (non-video)

### 1.7.1 What data can I access?

You can access: - response data from responses for which you have confirmed consent in the Consent Manager - account, demographic, and child data from those responses: you will see these accounts under 'Manage Participants'; if some siblings but not others have participated in one of your studies and you have confirmed consent, you will only see those siblings.

### 1.7.2 Accessing experiment data

You can see and download collected responses either via the Lookit experimenter interface or using the API.

A researcher with edit permissions for a particular study can download session data in JSON or CSV format via the Experimenter interface. A session record in a Postgres database is created each time a participant starts the study, and includes a timestamp, account information, condition assignment, the sequence of frames the participant actually saw, and frame-specific information for each frame (included in an 'expData' structure which is a JSON object with keys corresponding to frame nicknames as defined in the study definition JSON). Each frame type may save different data, e.g. form responses; frames that record webcam video include the video filename(s). The data captured by a particular frame are listed in the frame documentation at <http://lookit.github.io/ember-lookit-frameplayer>, under 'Methods' > 'serializeContent'. Additionally, event data is captured for each frame and included under an eventTimings key within the frame data JSON, minimally including a timestamped event when the user proceeds to the next frame. These events are listed under 'Events' in the documentation.

### 1.7.3 Structure of session data

The data saved when a subject participates in a study varies based on how that experiment is defined. For concreteness, let's start by looking at an example of the data you can download about a single session. (The eventTimings objects have been shortened to show just a single event.)

```
{
  "response": {
    "id": 1190,
    "uuid": "d96b3ba5-6806-4c09-86e2-77456163eb5a",
    "sequence": [
      "0-video-config",
      "1-video-consent",
      "2-instructions",
      "3-mood-survey",
      "4-pref-phys-videos",
      "5-exit-survey"
    ],
    "conditions": {
      "4-pref-phys-videos": {
        "showStay": 18,
        "startType": 21
      }
    },
    "exp_data": {
```

(continues on next page)



(continued from previous page)

```

    "3-mood-survey": {
      "active": "4",
      "rested": "1",
      "healthy": "2",
      "eventTimings": [
        {
          "eventType": "exp-mood-questionnaire:nextFrame",
          "timestamp": "2018-07-06T23:56:06.459Z"
        }
      ]
    },
    "0-video-config": {
      "eventTimings": [
        {
          "pipeId": "",
          "videoId": "videoStream_0f620873-2847-4eeb-9854-df7898934c17_
↪0-video-config_d96b3ba5-6806-4c09-86e2-77456163eb5a_1530921299537_405",
          "eventType": "exp-video-config:recorderReady",
          "timestamp": "2018-07-06T23:54:59.548Z",
          "streamTime": null
        }
      ]
    },
    "2-instructions": {
      "eventTimings": [
        {
          "eventType": "exp-physics-intro:nextFrame",
          "timestamp": "2018-07-06T23:55:53.530Z"
        }
      ]
    },
    "1-video-consent": {
      "videoId": "videoStream_0f620873-2847-4eeb-9854-df7898934c17_1-video-
↪consent_d96b3ba5-6806-4c09-86e2-77456163eb5a_1530921346557_292",
      "videoList": [
        "videoStream_0f620873-2847-4eeb-9854-df7898934c17_1-video-consent_
↪d96b3ba5-6806-4c09-86e2-77456163eb5a_1530921346557_292"
      ],
      "eventTimings": [
        {
          "pipeId": "",
          "videoId": "videoStream_0f620873-2847-4eeb-9854-df7898934c17_
↪1-video-consent_d96b3ba5-6806-4c09-86e2-77456163eb5a_1530921346557_292",
          "eventType": "exp-video-consent:recorderReady",
          "timestamp": "2018-07-06T23:55:46.558Z",
          "streamTime": 0
        }
      ]
    },
    "5-exit-survey": {
      "feedback": "",
      "birthDate": "2018-07-03T04:00:00.000Z",
      "useOfMedia": "private",
      "withdrawal": false,
      "eventTimings": [
        {
          "eventType": "exp-exit-survey:nextFrame",

```

(continues on next page)

(continued from previous page)

```

        "timestamp": "2018-07-06T23:57:02.201Z"
      },
    ],
    "databraryShare": "no"
  },
  "4-pref-phys-videos": {
    "videoId": "videoStream_0f620873-2847-4eeb-9854-df7898934c17_4-pref-
    ↪phys-videos_d96b3ba5-6806-4c09-86e2-77456163eb5a_1530921371545_923",
    "videoList": [
      "videoStream_0f620873-2847-4eeb-9854-df7898934c17_4-pref-phys-
      ↪videos_d96b3ba5-6806-4c09-86e2-77456163eb5a_1530921371545_923"
    ],
    "videosShown": [
      "https://s3.amazonaws.com/lookitcontents/exp-physics-final/
      ↪stimuli/stay/webm/sbs_stay_near_mostly-on_book_c2_green_NN.webm",
      "https://s3.amazonaws.com/lookitcontents/exp-physics-final/
      ↪stimuli/stay/webm/sbs_stay_mostly-on_near_book_c2_green_NN.webm"
    ],
    "eventTimings": [
      {
        "pipeId": "",
        "videoId": "videoStream_0f620873-2847-4eeb-9854-df7898934c17_
        ↪4-pref-phys-videos_d96b3ba5-6806-4c09-86e2-77456163eb5a_1530921371545_923",
        "eventType": "exp-video-physics:recorderReady",
        "timestamp": "2018-07-06T23:56:11.549Z",
        "streamTime": 0
      }
    ]
  },
  },
  "global_event_timings": [],
  "completed": true
},
"study": {
  "id": 12,
  "uuid": "0f620873-2847-4eeb-9854-df7898934c17"
},
"participant": {
  "id": 3047,
  "uuid": "31692a6c-df1e-47e1-8ad0-e2780d095c05",
  "nickname": "Kim"
},
"child": {
  "id": 3749,
  "uuid": "470a0d33-77ee-4dd5-a64e-ec7231f23913",
  "name": "ExperimenterChild",
  "birthday": "2018-02-05",
  "gender": "f",
  "age_at_birth": "30",
  "additional_information": "Test child"
}
}

```

There are four top-level keys in this data: response, study, participant, and child. Study, participant, and child information should be fairly self-explanatory: which study does this response pertain to, which family account created the response, and which child was participating. (The child key `age_at_birth` refers to gestational age in weeks at birth.) The response data contains information concerning this particular session: when it happened, what

condition the child was assigned to, events that happened as the family proceeded through the study, etc. The response properties are described below:

- *id*: short unique ID for the response
- *uuid*: long unique ID for the response (should be used as primary identifier)
- *sequence*: The sequence of **frames** the subject actually saw (after running randomization, etc.). Does not include frames skipped if they left early. The frame names follow the pattern `<order>-<frame.id>`, where `<order>` is the order in the overall sequence where this **frame** appeared, and `<frame.id>` is the identifier of the frame as defined in the ‘frames’ property of the experiment structure.
- *conditions*: An object containing information about conditions to which the subject was assigned in any frames that do randomization (choice frames). Keys are in the format `<order>-<frame.id>` corresponds with the `<order>` from the ‘sequence’ of the *original* experiment structure, and the `<frame.id>` again corresponds with the identifier of the frame as defined in the ‘frames’ property of the experiment structure. Data will be stored in conditions for the *first* frame created by a randomizer (top-level only for now, i.e. not from nested randomizers). Values are objects containing mappings from condition names to their values for this session. The data stored by a particular randomizer can be found under `methods: conditions` in the [randomizer documentation](#)
- *global\_event\_timings*: A list of events recorded during the study, not tied to a particular frame. Currently used for recording early exit from the study; an example value is:

```
[
  {
    "exitType": "manualInterrupt",
    "eventType": "exitEarly",
    "timestamp": "2018-07-06T23:56:55.282Z",
    "lastPageSeen": 10
  }
]
```

- *completed*: A true/false flag indicating whether or not the participant submitted the last frame of the study. Note that this may not line up with your notion of whether the participant completed the study, in two ways: first, *completed* will be true even if the participant leaves early, as long as they submit the exit survey which they skip to when pressing F1. Second, *completed* will be false if they don’t submit that exit survey, even if they completed all of the important experimental parts of the study.
- *exp\_data*: A JSON object containing the data collected by each **frame** in the study. More on this to follow.

### 1.7.4 Interpreting `exp_data`

Here’s an example of data collected during a session (note: not all fields are shown):

```
{
  "sequence": [
    "0-intro-video",
    "1-survey",
    "2-exit-survey"
  ],
  "conditions": {
    "1-survey": {
      "parameterSet": {
        "QUESTION1": "What is your favorite color?",
        "QUESTION2": "What is your favorite number?"
      },

```

(continues on next page)

(continued from previous page)

```

        "conditionNum": 0
      },
    },
    "exp_data": {
      "0-intro-video": {
        "eventTimings": [{
          "eventType": "nextFrame",
          "timestamp": "2016-03-23T16:28:20.753Z"
        }]
      },
      "1-survey": {
        "formData": {
          "name": "Sam",
          "favPie": "pecan"
        },
        "eventTimings": [{
          "eventType": "nextFrame",
          "timestamp": "2016-03-23T16:28:26.925Z"
        }]
      },
      "2-exit-survey": {
        "formData": {
          "thoughts": "Great!",
          "wouldParticipateAgain": "Yes"
        },
        "eventTimings": [{
          "eventType": "nextFrame",
          "timestamp": "2016-03-23T16:28:32.339Z"
        }]
      }
    }
  }
}

```

`exp_data` is an object with three keys that correspond with the frame names from ‘sequence’. Each of the associated values has an `eventTimings` property. This is a place to collect user-interaction events during an experiment, and by default contains the ‘nextFrame’ event which records when the user progressed to the next **frame** in the ‘sequence’. You can see which events a particular frame records by looking at the ‘Events’ tab in its [frame documentation](#). Events recorded by a frame that does video recording will include additional information, for instance to indicate when relative to the video stream this event happened.

The other properties besides ‘eventTimings’ are dependent on the **frame** type. You can see which other properties a particular frame type records by looking at the parameters of the `serializeContent` method under the ‘Methods’ tab in its [frame documentation](#).

## 1.8 Consent manager

### 1.8.1 Overview

At the start of a Lookit study, the parent is asked to provide a verbal statement of informed consent. Unlike in the lab (or to a greater extent), it is technically possible for you to end up collecting data from a parent who did NOT consent to participate - e.g., someone idly clicking through who may not understand that this is a research study to do with a child.

For this reason it is critical that you confirm informed consent before using any data from a response! This is baked

into the Lookit experimenter interface: you actually do not receive access to responses, or to the associated child, account, or demographic data, until you confirm consent using the consent manager.

Responses submitted on Lookit start out with a consent status of 'Pending.' Then a researcher working on this study can either 'approve' or 'reject' the consent video.

## **1.8.2 Managing consent rulings**

From your study detail page, click 'View Responses' and you will be taken to the 'Consent manager' view.

Manage Studies / Example study / Consent Manager

## Consent Manager

Consent Manager Individual Responses All Responses Demographic Snapshots Attachments

Responses

Show Currently – Pending ▾

Responses

Fri March 22, 3:09 p.m. EDT  
No previous ruling.

Pending ▾

Revert to Pending 0

Approvals 0 Rejections 0

Submit Rulings & Comments ↗

Reset Current Choices ↻

Processed: 3/22/2019, 3:12:07 PM

0:09 / 0:09

Comment on session. These will be saved upon submit.

« 1 »

Response Statistics

Pending Responses 1

Accepted Responses 7

Unique Children 2

Rejected Responses 0

Children with no accepted responses 0

Total # Responses 8

Total # Children 3

### Session Data

#### General Information

ID	UUID	Sequence	Conditions	Global Event Timing	Completed	Withdrawn
568	f7da2bd3-fb9f-4fdc-a88e-587935813e39	0-video-config,1-video-consent,2-instructions,3-video-preview-exp,4-video-preview,5-example-survey,6-final-instructions,7-video-quality,8-pref-phys-videos,9-pref-phys-videos,10-pref-phys-videos,11-exit-survey,11-exit-survey	{"8-pref-phys-videos": [{"showStay": 12, "startType": 7, "whichObjects": [7, 78, 678, 2]}]}		true	null

#### Participant Information

ID	UUID	Nickname
77	2ee68087-6736-4c56-81e6-653da911d920	

#### Child Information

ID	UUID	Name	Birthday	Gender	Age at Birth	Additional Info
61	65d9879d-427f-4a68-b691-3b32ef12c2c2	Tiny Tim	2019-03-04	m	39	

#### Full Session Data

[Terms of Use](#) | [Privacy](#)

At the left, you will see a list of responses. By default the responses with ‘Pending’ consent status are displayed; you can use the dropdown menu to show ‘Accepted’ or ‘Rejected’ consent videos instead.

### 1.8.2.1 Making consent rulings

When you click on a response, any consent videos from that response are shown to the right. (It is possible, although rare, for there to be multiple consent videos associated with a single response; this will become more common when some researchers are collecting both parental consent and child assent, which would be judged together.) A minimal summary of the data is shown below so that you can see whether the child is in the age range for the study and how far the family got. Unless this response already has already been accepted, you will NOT see 'Full Session Data' shown, because this could include more sensitive information.

Watch the video, and decide whether it shows informed consent. You can choose to 'Accept' or 'Reject' a response, and can enter a comment if desired to keep track of any additional information. You can enter a comment without changing the consent ruling (e.g., to say "Emailed this family to confirm consent"). In general, you should 'accept' consent only when the consent video shows an adult reading the consent statement audibly (or signing in ASL), but see the [Terms of Use](#) for details (for instance, you may be able to contact a family to confirm consent by email in some cases).

Repeat for each consent video. When you are done for now, click 'Submit Rulings and Comments' to save your judgments. These changes will immediately be reflected in the number of responses available in the 'individual responses' and 'all responses' views, as well as with respect to demographic and participant data you have access to.

Consent rulings can be changed after an initial ruling is made; for instance, you can use the dropdown menu to display 'Accepted' responses and either 'Reject' or 'Revert to pending.'

The most recent consent ruling, the time of that ruling, any comment, and the name of the researcher who made the ruling, will be included in the JSON/CSV data for this response.

### 1.8.2.2 Response statistics

A summary of responses is shown to the right of the consent manager, providing some very basic information about the non-consented responses that may be useful for publication of results. You can see how many responses are still pending consent confirmation; how many accepted responses there are (from how many unique children); and how many responses were rejected (from how many unique children who did not also have some response accepted).

### 1.8.2.3 Withdrawn responses

If a parent chooses to withdraw video data at the end of the study, that will be noted in the list item for the response (before the comment it will say 'Withdrawn' and the response will be crossed out). All video data beyond consent will be inaccessible to researchers, and it will be deleted automatically from Lookit servers after seven days.

However, you are still able to make a consent ruling about the consent video; this will still impact access to the remaining non-video response data as well as associated child, demographic, and account data.

## 1.9 Using the API

### 1.9.1 What is the API for?

Using the Lookit API allows you to programatically retrieve or update data (other than video data), rather than manually downloading JSON or CSV files from the Experimenter site. It is also currently the only way to update feedback to participants, although a way to do that via the experimenter interface is coming soon!

Researchers do not in general need to use the API in order to use Lookit to run their studies, but it is available if needed.

## 1.9.2 API Tips

### 1.9.2.1 General

Most endpoints in this API are just meant for retrieving data. Typically, you can retrieve data associated with studies you have permission to view, or view any data that belongs to you. You can only create *responses* and *feedback* through the API. You can only update *responses* and *feedback* through the API. There is *nothing* that is permitted to be deleted through the API. For a set of sample functions using the API from Python, please see <https://github.com/kimberscott/lookit-data-processing/blob/coding-workflow-multilab/scripts/experimenter.py>

### 1.9.2.2 API Formatting

This API generally conforms to the [JSON-API 1.0 spec](#). Top-level keys are underscored, while nested key formatting will be the casing that is stored in the db. For example, in the study response below, top-level attributes like *exp\_data* and *global\_event\_timings* are underscored. However, nested keys like *5-5-mood-survey* and *napWakeUp* retain the casing given to them by the *exp-player*.

```
{
  "type": "responses",
  "id": "bdebd15b-adc7-4377-b2f6-e9f3de70dd19",
  "attributes": {
    "conditions": {
      "8-pref-phys-videos": {
        "showStay": 8,
        "startType": 5,
        "whichObjects": [
          8,
          5,
          6,
          8
        ]
      }
    },
    "global_event_timings": [
      {
        "exitType": "browserNavigationAttempt",
        "eventType": "exitEarly",
        "timestamp": "2017-10-31T20:30:38.514Z",
        "lastPageSeen": 6
      }
    ],
    "exp_data": {
      "5-5-mood-survey": {
        "active": "1",
        "rested": "1",
        "healthy": "1",
        "lastEat": "6:00",
        "energetic": "1",
        "napWakeUp": "11:00",
        "childHappy": "1",
        "doingBefore": "s",
        "parentHappy": "1",
        "eventTimings": [
          {
            "eventType": "nextFrame",
            "timestamp": "2017-10-31T20:10:17.269Z"
          }
        ]
      }
    }
  }
}
```

(continues on next page)



(continued from previous page)

```

        },
        "ontopofstuff": "1",
        "usualNapSchedule": "no"
    },
    "sequence": [
        "5-5-mood-survey"
    ],
    "completed": false
},
"relationships": {
    "child": {
        "links": {
            "related": "http://localhost:8000/api/v1/children/da27faf2-c3d2-4701-
↪b3bb-dd865f89c1a1/"
        }
    },
    "study": {
        "links": {
            "related": "http://localhost:8000/api/v1/studies/e729321f-418f-4728-
↪992c-9364623dbe9b/"
        }
    },
    "demographic_snapshot": {
        "links": {
            "related": "http://localhost:8000/api/v1/demographics/341ea7c7-f657-
↪4ab2-a530-21ac293e7d6f/"
        }
    }
},
"links": {
    "self": "http://localhost:8000/api/v1/responses/bdebd15b-adc7-4377-b2f6-
↪e9f3de70dd19/"
}
}

```

### 1.9.2.3 Content-Type

The following Content-Type must be in the header of the request: *application/vnd.api+json*.

### 1.9.2.4 Authentication



We are using a token-based HTTP Authentication scheme.

- Go to Experimenter's admin app to create a token */admin/authtoken/token/add/* (Only users marked as “Staff” can access the admin app; for now please ask Kim to provide you with a token.)

Home › Auth Token › Tokens › Add Token

## Add Token

User:

Save and add another

Save and continue editing

SAVE

- Select your user from the dropdown and hit ‘Save’. Copy the token.

Home › Auth Token › Tokens

Select Token to change

ADD TOKEN +

Action:  Go 0 of 1 selected

<input type="checkbox"/>	KEY	USER	CREATED
<input type="checkbox"/>	123456789abcdefghijklmnopqrstuvwxyz	<User: Test User>	Sept. 6, 2017, 5:20 p.m.

1 Token

- Include this token in your Authorization HTTP header. The word “Token” should come before it.

```
curl -X GET <API_URL_HERE> -H 'Authorization: Token <paste_token_here>'
```

- For example, here’s how you would access users using curl:

```
curl -X GET https://localhost:8000/api/v1/users/ -H 'Authorization: Token_
↪123456789abcdefghijklmnopqrstuvwxyz'
```

- Here is an example of a POST request using curl, note the presence of the content-type header as well as the authorization header:

```
curl -X POST http://localhost:8000/api/v1/feedback/ -H "Content-Type: application/
↪vnd.api+json" -H 'Authorization: Token abcdefghijklmnopqrstuvwxyzyour-token-here' -
↪d '{"data": {"attributes": {"comment": "Test comment"}, "relationships": {"response
↪": {"data": {"type": "responses", "id": "91c15b81-bb25-437a-8299-13cf4c83fed6"}}},
↪"type": "feedback"}}'
```

### 1.9.2.5 Pagination

- This API is paginated, so results are returned in batches of 10. Follow the pagination links in the API response to fetch the subsequent pages of data. In the example below, the “links” section of the API response has the first, last, next, and previous links.

*Sample Response:*

```
{
  "links": {
    "first": "http://localhost:8000/api/v1/responses/?page=1",
    "last": "http://localhost:8000/api/v1/responses/?page=5",
    "next": "http://localhost:8000/api/v1/responses/?page=2",
    "prev": null,
    "meta": {
      "page": 1,
```

(continues on next page)

(continued from previous page)

```

    "pages": 5,
    "count": 50
  }
}

```

## 1.9.3 Available Endpoints

### 1.9.3.1 Children

#### Viewing the list of children

GET /api/v1/children/

Permissions: Must be authenticated. You can only view children that have responded to studies you have permission to view, or your own children. Users with *can\_read\_all\_user\_data* permissions can view all children of active users in the database via this endpoint.

Ordering: Children can be sorted by birthday using the *ordering* query parameter. For example, to sort oldest to youngest:

GET <http://localhost:8000/api/v1/children/?ordering=birthday>

Add a '-' before birthday to sort youngest to oldest:

GET <http://localhost:8000/api/v1/children/?ordering=-birthday>

*Sample Response:*

```

{
  "links": {
    "first": "http://localhost:8000/api/v1/children/?page=1",
    "last": "http://localhost:8000/api/v1/children/?page=1",
    "next": null,
    "prev": null,
    "meta": {
      "page": 1,
      "pages": 1,
      "count": 1
    }
  },
  "data": [
    {
      "type": "children",
      "id": "0b380366-31b9-45c1-86ef-0fd9ea238ff4",
      "attributes": {
        "given_name": "Ashley",
        "birthday": "2015-01-01",
        "gender": "f",
        "age_at_birth": "36",
        "additional_information": "",
        "deleted": false
      },
      "relationships": {
        "user": {
          "links": {

```

(continues on next page)

(continued from previous page)

```
      "related": "http://localhost:8000/api/v1/users/834bbf33-b249-
↪4737-a041-43574cd137a7/"
    }
  },
  "links": {
    "self": "http://localhost:8000/api/v1/children/0b380366-31b9-45c1-
↪86ef-0fd9ea238ff4/"
  }
}
]
```

## Retrieving a single child

GET /api/v1/children/<child\_id>/

Permissions: Must be authenticated. You can only view a child if he or she has responded to a study you have permission to view. You can additionally view your own child via the API.

*Sample Response:*

```
{
  "data": {
    "type": "children",
    "id": "0b380366-31b9-45c1-86ef-0fd9ea238ff4",
    "attributes": {
      "given_name": "Ashley",
      "birthday": "2015-01-01",
      "gender": "f",
      "age_at_birth": "36",
      "additional_information": "",
      "deleted": false
    },
    "relationships": {
      "user": {
        "links": {
          "related": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-
↪a041-43574cd137a7/"
        }
      },
      "links": {
        "self": "http://localhost:8000/api/v1/children/0b380366-31b9-45c1-86ef-
↪0fd9ea238ff4/"
      }
    }
  }
}
```

## Creating a Child

POST /api/v1/children/

METHOD NOT ALLOWED. Not permitted via the API.

## Updating a Child.

PUT /api/v1/children/<child\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

## Deleting a Child

DELETE /api/v1/children/<child\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

### 1.9.3.2 Demographic Data

#### Viewing the list of demographic data

GET /api/v1/demographics/

Permissions: Must be authenticated. You can only view demographics of participants whose children have responded to studies you can view. You can additionally view your own demographic data via the API. Users with *can\_read\_all\_user\_data* permissions can view all demographics of active users in the database via this endpoint.

*Sample Response:*

```
{
  "links": {
    "first": "http://localhost:8000/api/v1/demographics/?page=1",
    "last": "http://localhost:8000/api/v1/demographics/?page=1",
    "next": null,
    "prev": null,
    "meta": {
      "page": 1,
      "pages": 1,
      "count": 1
    }
  },
  "data": [
    {
      "type": "demographics",
      "id": "f5fa60ca-d428-46cd-9820-846492dd9900",
      "attributes": {
        "number_of_children": "1",
        "child_birthdays": [
          "2015-01-01"
        ],
        "languages_spoken_at_home": "English and French",
        "number_of_guardians": "2",
        "number_of_guardians_explanation": "",
        "race_identification": [
          "white"
        ],
        "age": "30-34",
        "gender": "f",
        "education_level": "grad",
        "spouse_education_level": "bach",
        "annual_income": "30000",

```

(continues on next page)

(continued from previous page)

```

        "number_of_books": 100,
        "additional_comments": "",
        "country": "US",
        "state": "AZ",
        "density": "urban",
        "extra": {
            "no": "extra"
        }
    },
    "links": {
        "self": "http://localhost:8000/api/v1/demographics/f5fa60ca-d428-46cd-
↪9820-846492dd9900/"
    }
}
]
}

```

## Retrieving a single piece of demographic data

GET /api/v1/demographics/<demographic\_data\_id>/

Permissions: Must be authenticated. You can only view demographics of participants whose children have responded to studies you can view. You can additionally view your own demographic data via the API.

*Sample Response:*

```

{
  "data": {
    "type": "demographics",
    "id": "f5fa60ca-d428-46cd-9820-846492dd9900",
    "attributes": {
      "number_of_children": "1",
      "child_birthdays": [
        "2015-01-01"
      ],
      "languages_spoken_at_home": "English and French",
      "number_of_guardians": "2",
      "number_of_guardians_explanation": "",
      "race_identification": [
        "white"
      ],
      "age": "30-34",
      "gender": "f",
      "education_level": "grad",
      "spouse_education_level": "bach",
      "annual_income": "30000",
      "number_of_books": 100,
      "additional_comments": "",
      "country": "US",
      "state": "AZ",
      "density": "urban",
      "extra": {
        "no": "extra"
      }
    }
  },
}

```

(continues on next page)

(continued from previous page)

```

    "links": {
      "self": "http://localhost:8000/api/v1/demographics/f5fa60ca-d428-46cd-
↪9820-846492dd9900/"
    }
  }
}

```

## Creating Demographics

POST /api/v1/demographics/

METHOD NOT ALLOWED. Not permitted via the API.

## Updating Demographics

PUT /api/v1/demographics/<demographic\_data\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

## Deleting Demographics

DELETE /api/v1/demographics/<demographic\_data\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

### 1.9.3.3 Feedback

#### Viewing the list of feedback

GET /api/v1/feedback/

Permissions: Must be authenticated. You can only view feedback on study responses you have permission to view. Additionally, you can view feedback left on your own responses.

*Sample Response:*

```

{
  "links": {
    "first": "http://localhost:8000/api/v1/feedback/?page=1",
    "last": "http://localhost:8000/api/v1/feedback/?page=1",
    "next": null,
    "prev": null,
    "meta": {
      "page": 1,
      "pages": 1,
      "count": 1
    }
  },
  "data": [
    {
      "type": "feedback",
      "id": "cbfc64ee-30a3-491e-bd0e-1bef81540ea5",

```

(continues on next page)

(continued from previous page)

```

      "attributes": {
        "comment": "Thanks for participating!  Next time, please center the_
↪webcam; you were off-center in many of the video clips."
      },
      "relationships": {
        "response": {
          "links": {
            "related": "http://localhost:8000/api/v1/responses/841c8a77-
↪b322-4e25-8e03-47a83fa326ff/"
          }
        },
        "researcher": {
          "links": {
            "related": "http://localhost:8000/api/v1/users/834bbf33-b249-
↪4737-a041-43574cd137a7/"
          }
        }
      },
      "links": {
        "self": "http://localhost:8000/api/v1/feedback/cbfc64ee-30a3-491e-
↪bd0e-1bef81540ea5/"
      }
    }
  ]
}

```

## Retrieving a single piece of feedback

GET /api/v1/feedback/<feedback\_id>/

Permissions: Must be authenticated. You can only retrieve feedback attached to a study response you have permission to view. Additionally, you can retrieve feedback attached to one of your own responses.

*Sample Response:*

```

{
  "data": {
    "type": "feedback",
    "id": "cbfc64ee-30a3-491e-bd0e-1bef81540ea5",
    "attributes": {
      "comment": "Thanks for participating!  Next time, please center the_
↪webcam; you were off-center in many of the video clips."
    },
    "relationships": {
      "response": {
        "links": {
          "related": "http://localhost:8000/api/v1/responses/841c8a77-b322-
↪4e25-8e03-47a83fa326ff/"
        }
      },
      "researcher": {
        "links": {
          "related": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-
↪a041-43574cd137a7/"
        }
      }
    }
  }
}

```

(continues on next page)



(continued from previous page)

```

    }
  },
  "links": {
    "self": "http://localhost:8000/api/v1/feedback/cbfc64ee-30a3-491e-bd0e-
↪1bef81540ea5/"
  }
}
}

```

## Creating Feedback

POST /api/v1/feedback/

Permissions: Must be authenticated. Must have permission to edit the study response where you are leaving feedback (which only admins have).

*Sample Request body:*

```

{
  "data": {
    "attributes": {
      "comment": "Thank you so much for participating in round one! Please try to
↪respond to the second round some time in the next three weeks!"
    },
    "relationships": {
      "response": {
        "data": {
          "type": "responses",
          "id": "841c8a77-b322-4e25-8e03-47a83fa326ff"
        }
      }
    },
    "type": "feedback"
  }
}

```

*Sample Response*

```

{
  "data": {
    "type": "feedback",
    "id": "aabf86c7-3dc0-4284-844c-89e04a1f154f",
    "attributes": {
      "comment": "Thank you so much for participating in round one! Please try
↪to respond to the second round some time in the next three weeks!"
    },
    "relationships": {
      "response": {
        "links": {
          "related": "http://localhost:8000/api/v1/responses/841c8a77-b322-
↪4e25-8e03-47a83fa326ff/"
        }
      },
      "researcher": {
        "links": {

```

(continues on next page)

(continued from previous page)

```
      "related": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-
↪a041-43574cd137a7/"
    },
    },
    "links": {
      "self": "http://localhost:8000/api/v1/feedback/aabf86c7-3dc0-4284-844c-
↪89e04a1f154f/"
    }
  }
}
```

## Updating Feedback

PATCH /api/v1/feedback/<feedback\_id>/

Permissions: Must be authenticated. Must have permission to edit the study response where you are changing feedback (which only admins have).

*Sample Request body:*

```
{
  "data": {
    "attributes": {
      "comment": "Changed comment"
    },
    "type": "feedback",
    "id": "ebf41029-02d7-49f5-8adb-1e32d4ac22a5"
  }
}
```

## Deleting Feedback

DELETE /api/v1/feedback/<feedback\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

### 1.9.3.4 Organizations

#### Viewing the list of organizations

GET /api/v1/organizations/

Permissions: Must be authenticated.

*Sample Response:*

```
{
  "links": {
    "first": "http://localhost:8000/api/v1/organizations/?page=1",
    "last": "http://localhost:8000/api/v1/organizations/?page=1",
    "next": null,
    "prev": null,
  }
}
```

(continues on next page)

(continued from previous page)

```

    "meta": {
      "page": 1,
      "pages": 1,
      "count": 1
    },
    "data": [
      {
        "type": "organizations",
        "id": "665c4457-a02e-4842-bd72-7043de3d66d0",
        "attributes": {
          "name": "MIT"
        },
        "links": {
          "self": "http://localhost:8000/api/v1/organizations/665c4457-a02e-
↪4842-bd72-7043de3d66d0/"
        }
      }
    ]
  }

```

## Retrieving a single organization

GET /api/v1/organizations/<organization\_id>/

Permissions: Must be authenticated.

*Sample Response:*

```

{
  "data": {
    "type": "organizations",
    "id": "665c4457-a02e-4842-bd72-7043de3d66d0",
    "attributes": {
      "name": "MIT"
    },
    "links": {
      "self": "http://localhost:8000/api/v1/organizations/665c4457-a02e-4842-
↪bd72-7043de3d66d0/"
    }
  }
}

```

## Creating an Organization

POST /api/v1/organizations/

METHOD NOT ALLOWED. Not permitted via the API.

## Updating an Organization

PUT /api/v1/organizations/<organization\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

## Deleting an Organization

DELETE /api/v1/organizations/<organization\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

### 1.9.3.5 Responses

#### Viewing the list of responses

GET /api/v1/responses/

Permissions: Must be authenticated. You can only view responses to studies you have permission to view. Additionally, you can view your own responses through the API.

Sort Order: By default, responses are sorted reverse date\_modified, meaning the most recently modified responses appear first.

*Sample Response:*

```
{
  "links": {
    "first": "http://localhost:8000/api/v1/feedback/?page=1",
    "last": "http://localhost:8000/api/v1/feedback/?page=1",
    "next": null,
    "prev": null,
    "meta": {
      "page": 1,
      "pages": 1,
      "count": 1
    }
  },
  "data": [
    {
      "type": "responses",
      "id": "8260ca67-6ec0-4749-ba11-fa35612ea030",
      "attributes": {
        "conditions": {

        },
        "global_event_timings": [
          {
            "exit_type": "browserNavigationAttempt",
            "timestamp": "2017-09-05T14:33:41.322Z",
            "event_type": "exitEarly",
            "last_page_seen": 0
          }
        ],
        "exp_data": {

        },
        "sequence": [

        ],
        "completed": false
      },
      "relationships": {
```

(continues on next page)

(continued from previous page)

```

    "child": {
      "links": {
        "related": "http://localhost:8000/api/v1/children/0b380366-31b9-45c1-
↪86ef-0fd9ea238ff4/"
      }
    },
    "study": {
      "links": {
        "related": "http://localhost:8000/api/v1/studies/a8a80880-5539-4650-
↪9387-c62afa202d43/"
      }
    },
    "demographic_snapshot": {
      "links": {
        "related": "http://localhost:8000/api/v1/demographics/f5fa60ca-d428-
↪46cd-9820-846492dd9900/"
      }
    }
  },
  "links": {
    "self": "http://localhost:8000/api/v1/responses/8260ca67-6ec0-4749-ba11-
↪fa35612ea030/"
  }
}
]
}

```

## Retrieving a single response

GET /api/v1/responses/<response\_id>/

Permissions: Must be authenticated. You can only view responses to studies you have permission to view as well as your own responses.

*Sample Response:*

```

{
  "data": {
    "type": "responses",
    "id": "8260ca67-6ec0-4749-ba11-fa35612ea030",
    "attributes": {
      "conditions": {},
      "global_event_timings": [
        {
          "exit_type": "browserNavigationAttempt",
          "timestamp": "2017-09-05T14:33:41.322Z",
          "event_type": "exitEarly",
          "last_page_seen": 0
        }
      ],
      "exp_data": {},
      "sequence": [],
      "completed": false
    },
    "relationships": {

```

(continues on next page)

(continued from previous page)

```
      "child": {
        "links": {
          "related": "http://localhost:8000/api/v1/children/0b380366-31b9-
↪45c1-86ef-0fd9ea238ff4/"
        }
      },
      "study": {
        "links": {
          "related": "http://localhost:8000/api/v1/studies/a8a80880-5539-
↪4650-9387-c62afa202d43/"
        }
      },
      "demographic_snapshot": {
        "links": {
          "related": "http://localhost:8000/api/v1/demographics/f5fa60ca-
↪d428-46cd-9820-846492dd9900/"
        }
      }
    },
    "links": {
      "self": "http://localhost:8000/api/v1/responses/8260ca67-6ec0-4749-ba11-
↪fa35612ea030/"
    }
  }
}
```

## Creating a Response

POST /api/v1/responses/. Possible to do programmatically, but really intended to be used by ember-lookit-frameplayer app.

Permissions: Must be authenticated. Child in response must be your child.

*Sample Request body:*

```
{
  "data": {
    "attributes": {},
    "relationships": {
      "child": {
        "data": {
          "type": "children",
          "id": "0b380366-31b9-45c1-86ef-0fd9ea238ff4"
        }
      },
      "study": {
        "data": {
          "type": "studies",
          "id": "a8a80880-5539-4650-9387-c62afa202d43"
        }
      }
    }
  },
  "type": "responses"
}
```

## Updating a Response

PATCH /api/v1/responses/<response\_id>/ Possible to do programmatically, but really intended for the ember-lookit-frameplayer to update as it moves through each frame of the study.

*Sample Request body:*

```
{
  "data": {
    "attributes": {
      "conditions": {"cloudy": "skies"}
    },
    "type": "responses",
    "id": "51c0a355-375d-481f-a3d0-6471db8f9f14"
  }
}
```

## Deleting a Response

DELETE /api/v1/responses/<response\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

### 1.9.3.6 Studies

#### Viewing the list of studies

GET /api/v1/studies/

Permissions: Must be authenticated. You can view studies that are active/public as well as studies you have permission to edit.

Sort Order: By default, studies are sorted reverse date\_modified, meaning the most recently modified studies appear first.

*Sample Response:*

```
{
  "links": {
    "first": "http://localhost:8000/api/v1/studies/?page=1",
    "last": "http://localhost:8000/api/v1/studies/?page=1",
    "next": null,
    "prev": null,
    "meta": {
      "page": 1,
      "pages": 1,
      "count": 1
    }
  },
  "data": [
    {
      "type": "studies",
      "id": "65680ade-510c-4437-a58a-e41d4b94d8ed",
      "attributes": {
        "name": "Sample Study",

```

(continues on next page)

(continued from previous page)

```

        "date_modified": "2017-09-06T19:33:24.826892Z",
        "short_description": "A short description of your study would go here.",
        "long_description": "A longer purpose of your study would be here.",
        "criteria": "Children should be around five.",
        "duration": "20 minutes",
        "contact_info": "Contact Sally",
        "image": "http://localhost:8000/media/study_images/download.jpeg",
        "structure": {
            "frames": {},
            "sequence": []
        },
        "display_full_screen": true,
        "exit_url": "http://www.cos.io",
        "state": "created",
        "public": true
    },
    "relationships": {
        "organization": {
            "links": {
                "related": "http://localhost:8000/api/v1/organizations/
↪665c4457-a02e-4842-bd72-7043de3d66d0/"
            }
        },
        "creator": {
            "links": {
                "related": "http://localhost:8000/api/v1/users/834bbf33-b249-
↪4737-a041-43574cd137a7/"
            }
        },
        "responses": {
            "links": {
                "related": "http://localhost:8000/api/v1/studies/65680ade-
↪510c-4437-a58a-e41d4b94d8ed/responses/"
            }
        },
        "links": {
            "self": "http://localhost:8000/api/v1/studies/65680ade-510c-4437-a58a-
↪e41d4b94d8ed/"
        }
    }
}

```

## Retrieving a single study

GET /api/v1/studies/<study\_id>/

Permissions: Must be authenticated. You can fetch an active study or a study you have permission to edit.

*Sample Response:*

```

{
  "data": {

```

(continues on next page)



(continued from previous page)

```

    "type": "studies",
    "id": "65680ade-510c-4437-a58a-e41d4b94d8ed",
    "attributes": {
      "name": "Sample Study",
      "date_modified": "2017-09-06T19:33:24.826892Z",
      "short_description": "A short description of your study would go here.",
      "long_description": "A longer purpose of your study would be here.",
      "criteria": "Children should be around five.",
      "duration": "20 minutes",
      "contact_info": "Contact Sally",
      "image": "http://localhost:8000/media/study_images/download.jpeg",
      "structure": {
        "frames": {},
        "sequence": []
      },
      "display_full_screen": true,
      "exit_url": "http://www.cos.io",
      "state": "created",
      "public": true
    },
    "relationships": {
      "organization": {
        "links": {
          "related": "http://localhost:8000/api/v1/organizations/665c4457-
↪a02e-4842-bd72-7043de3d66d0/"
        }
      },
      "creator": {
        "links": {
          "related": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-
↪a041-43574cd137a7/"
        }
      },
      "responses": {
        "links": {
          "related": "http://localhost:8000/api/v1/studies/65680ade-510c-
↪4437-a58a-e41d4b94d8ed/responses/"
        }
      },
      "links": {
        "self": "http://localhost:8000/api/v1/studies/65680ade-510c-4437-a58a-
↪e41d4b94d8ed/"
      }
    }
  }
}

```

## Retrieving a Study's responses

GET /api/v1/studies/<study\_id>/responses/

Permissions: Must be authenticated. Must have permission to view the responses to the particular study.

## Creating a Study

POST /api/v1/studies/

METHOD NOT ALLOWED. Not permitted via the API.

## Updating a Study

PUT /api/v1/studies/<study\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

## Deleting a Study

DELETE /api/v1/studies/<study\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

### 1.9.3.7 Users

#### Viewing the list of users

GET /api/v1/users/

Permissions: Must be authenticated. You can view participants that have responded to studies you have permission to view, as well as own user information. Endpoint can return both participants and researchers, if you have permission to view them. Users with *can\_read\_all\_user\_data* permissions can view all active users in the database via this endpoint. Usernames are only shown if user has *can\_read\_usernames* permissions.

*Sample Response:*

```
{
  "links": {
    "first": "http://localhost:8000/api/v1/users/?page=1",
    "last": "http://localhost:8000/api/v1/users/?page=1",
    "next": null,
    "prev": null,
    "meta": {
      "page": 1,
      "pages": 1,
      "count": 1
    }
  },
  "data": [
    {
      "type": "users",
      "id": "834bbf33-b249-4737-a041-43574cd137a7",
      "attributes": {
        "given_name": "Test",
        "middle_name": "",
        "family_name": "User",
        "identicon": "data:image/png;base64,aaaabbbbccccddddeeeeffffggggg",
        "is_active": true,
        "is_staff": true
      }
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    },
    "relationships": {
      "demographics": {
        "links": {
          "related": "http://localhost:8000/api/v1/users/834bbf33-b249-
↪4737-a041-43574cd137a7/demographics/"
        }
      },
      "organization": {
        "links": {
          "related": "http://localhost:8000/api/v1/organizations/
↪665c4457-a02e-4842-bd72-7043de3d66d0/"
        }
      },
      "children": {
        "links": {
          "related": "http://localhost:8000/api/v1/users/834bbf33-b249-
↪4737-a041-43574cd137a7/children/"
        }
      }
    },
    "links": {
      "self": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-a041-
↪43574cd137a7/"
    }
  }
]
}

```

## Retrieving a single user

GET /api/v1/users/<user\_id>/

Permissions: Must be authenticated. You can view participants that have responded to studies you have permission to view, as well as own user information.

*Sample Response:*

```

{
  "data": {
    "type": "users",
    "id": "834bbf33-b249-4737-a041-43574cd137a7",
    "attributes": {
      "given_name": "Test",
      "middle_name": "",
      "family_name": "User",
      "identicon": "data:image/png;base64,aaaabbbbccccddddeeeefffffgggg",
      "is_active": true,
      "is_staff": true
    },
    "relationships": {
      "demographics": {
        "links": {
          "related": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-
↪a041-43574cd137a7/demographics/"
        }
      }
    }
  }
}

```

(continues on next page)

(continued from previous page)

```
        }
      },
      "organization": {
        "links": {
          "related": "http://localhost:8000/api/v1/organizations/665c4457-
↪a02e-4842-bd72-7043de3d66d0/"
        }
      },
      "children": {
        "links": {
          "related": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-
↪a041-43574cd137a7/children/"
        }
      }
    },
    "links": {
      "self": "http://localhost:8000/api/v1/users/834bbf33-b249-4737-a041-
↪43574cd137a7/"
    }
  }
}
```

## Creating a User

POST /api/v1/users/

METHOD NOT ALLOWED. Not permitted via the API.

## Updating a User

PUT /api/v1/users/<user\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

## Deleting a User

DELETE /api/v1/users/<user\_id>/

METHOD NOT ALLOWED. Not permitted via the API.

---

### Developing new frames

---

## 2.1 Setup for custom frame development

Suppose that for your study, you need a frame that's not part of the standard `ember-lookit-frameplayer` library. Maybe you want to use a particular game you've already implemented in Javascript, or you want to slightly change how one of the existing frames works, or you want to hard-code a particular complicated counterbalancing scheme. That's okay! You can add a new frame to your own version of the `ember-lookit-frameplayer` repository, and tell Experimenter to use your Github fork of `ember-lookit-frameplayer` when building your study. But for efficiency, you will probably want to run Lookit on your own computer as you implement your new frame, so that you can test out changes immediately rather than repeatedly pushing your changes to Github and re-building your study on Experimenter. These instructions will walk you through setting up to run Lookit locally.

### 2.1.1 Overview

Even though we will probably just be changing the frame definitions in `ember-lookit-frameplayer`, we will need to install *both* the Django app (`lookit-api`) and the Ember app (`ember-lookit-frameplayer`), tell them how to talk to each other, and run both of those servers locally. In Experimenter, we need to add an organization to our superuser, and then add a child and demographic data. We then create a study locally. The `exp-player` needs to be linked for local development, and a token added to the headers of the API requests the `ember-lookit-frameplayer` is sending. We can then navigate directly to the study from the ember app to bypass the build process locally. This will enable you to make changes to frames locally and rapidly see the results of those changes, participating in a study just as if you were a participant on the Lookit website.

### 2.1.2 Django App steps

1. Follow the instructions to install the `django` app locally. Run the server.
2. Navigate to <http://localhost:8000/admin/> to login to Experimenter's admin app. You should be redirected to login. Use the superuser credentials created in the django installation steps.

3. Once you are in the Admin App, navigate to users, and then select your superuser. If you just created your django app, there should be two users to pick from, your superuser, and an anonymous user. In that case, your superuser information is here <http://localhost:8000/admin/accounts/user/2/change/>.
4. Update your superuser information through the admin app. We primarily need to add an organization to the user, but have to fill out the bold fields additionally in order to save the user information.
  - Family Name: *Your last name*
  - Organization: *Select MIT in dropdown. If no organizations are in the dropdown, create one through the admin app, and come back and add it here.*
  - Identicon: *If no identicon, just type random text here*
  - Timezone: *America/New\_York, as an example*
  - Locale: *en\_US, as an example*
  - Place a check in the checkbox by “Is Researcher”

Click “Save”.

5. Create a token to allow the Ember app to access the API by navigating to <http://localhost:8000/admin/authtoken/token/>. Click “Add Token”, find your superuser in the dropdown, and then click “Save”. You will need this token later.
6. Create a study by navigating to <http://localhost:8000/exp/studies/create/>. Fill out all the fields. The most important field is the `structure`, where you define the frames and the sequence of the frames. Be sure the frame and the details for the frame you are testing are listed in the structure.
7. Add demographic information to your superuser (just for testing purposes), so your superuser can participate in studies. Navigate to <http://localhost:8000/account/demographics/>. Scroll down to the bottom and hit “Save”. You’re not required to answer any questions, but hitting save will save a blank demographic data version for your superuser.
8. Create a child by navigating to <http://localhost:8000/account/children/>, and clicking “Add Child”. Fill out all the information with test data and click “Add child”.

Now we have a superuser with an organization, that has attached demographic data, and a child. We’ve created a study, as well as a token for accessing the API. Leave the django server running and switch to a new tab in your console.

Remember: The OAuth authentication used for access to Experimenter does not work when running locally. You can access Experimenter by first logging in as your superuser, or by giving another local user researcher permissions using the Admin app.

## 2.1.3 Ember App steps

1. Follow the instructions to install the [ember app](#) locally.
2. If you make changes to the frames, you should see notifications that files have changed in the console where your ember server is running, like this:

```
file changed components/exp-video-config/template.hbs
```

3. Add your token and lookit-api local host address to the `ember-lookit-frameplayer/.env` file. This will allow your Ember app to talk to your local API. Your `.env` file will now look like this:

```
PIPE_ACCOUNT_HASH=<account hash here>
PIPE_ENVIRONMENT=<environment here>
LOOKIT_API_KEY='Token <token here>'
LOOKIT_API_HOST='http://localhost:8000'
```

- If you want to use the HTML5 video recorder, you'll need to set up to use https locally. Open `ember-lookit-frameplayer/.ember-cli` and make sure it includes `ssl: true`:

```
"disableAnalytics": false,
"ssl": true
```

Create `server.key` and `server.crt` files in the root `ember-lookit-frameplayer` directory as follows:

```
openssl genrsa -des3 -passout pass:x -out server.pass.key 2048
openssl rsa -passin pass:x -in server.pass.key -out server.key
rm server.pass.key
openssl req -new -key server.key -out server.csr
openssl x509 -req -sha256 -days 365 -in server.csr -signkey server.key -out
server.crt
```

Leave the challenge password blank and enter `localhost` as the Common Name.

- Run the ember server: `ember serve`

## 2.1.4 Starting up once initial setup is completed

This is much quicker! Once you have gotten through the initial setup steps, you don't need to go through them every time you want to work on something.

- Start the Django app:

```
$ cd lookit-api
$ source VENVNAME/bin/activate
$ python manage.py runserver
```

- Start the Ember app:

```
$ cd ember-lookit-frameplayer
$ ember serve
```

- Log in as your local superuser at <http://localhost:8000/admin/>

## 2.1.5 Previewing a study

When you are previewing a study, the responses to the study will not be saved. You will get an error at the end of the study about this - that's expected and not something to worry about. Video attachments will be saved, however, with an id of "PREVIEW\_DATA\_DISREGARD". You do not need to create demographic data, or a child, since this is just a preview. You just need a study to navigate to. The URL for previewing is `/exp/studies/study_uuid/preview/`.

To fetch the identifier of the study, you can use the API. To fetch studies, navigate to <http://localhost:8000/api/v1/studies>. Copy the id of the study you created earlier.

Now, you can navigate to [https://localhost:4200/exp/studies/study\\_id/preview](https://localhost:4200/exp/studies/study_id/preview), replacing `study_id` with the id you obtained from the API. (For simplicity, bookmark this link while you're working!)

## 2.1.6 Participating in a study

To participate in a study locally, you need demographic data and a child attached to the logged in user, as well as a study.

Responses are saved to your local server. The URL for participating is `studies/study_uuid/child_uuid`. To fetch studies, navigate to <http://localhost:8000/api/v1/studies/>. Copy the id of the study you created earlier. To fetch children, navigate to <http://localhost:8000/api/v1/children/>. Copy the id of your child.

Finally, to participate in a study, navigate to [https://localhost:4200/studies/study\\_id/child\\_id](https://localhost:4200/studies/study_id/child_id), replacing `study_id` and `child_id` with the ids you obtained from the API. (For simplicity, bookmark this link while you're working!)

### 2.1.7 Where does my video go?

If you have set up the Pipe recorder environment variables as described in [the installation instructions](#), video recorded during your local testing will go to Pipe and then to an S3 bucket for Lookit development video. Contact us for directions about accessing this bucket. [TODO: documentation on setting up access.]

### 2.1.8 Using https

You may need to adjust browser settings to allow using https with the self-signed certificate. For instance, in Chrome, set Camera and Microphone permissions at <chrome://settings/content/siteDetails?site=https://localhost:4200>.

If not using https locally, replace the <https://localhost:4200> addresses with <http://localhost:4200>.

### 2.1.9 Further Reading / Useful Links

- <http://emberjs.com/>
- <http://ember-cli.com/>
- Development Browser Extensions - <https://chrome.google.com/webstore/detail/ember-inspector/bmdblncegkenkacieihfhpfjpoconhi> - <https://addons.mozilla.org/en-US/firefox/addon/ember-inspector/>

## 2.2 Creating custom frames

### 2.2.1 Overview

You may find you have a need for some experimental component not already included in Lookit. The goal of this section is to walk through extending the base functionality with your own code.

We use the term ‘frame’ to describe the combination of JavaScript file and Handlebars HTML template that compose a **block** of an experiment (see “Building your experiment”).

Experimenter is composed of two main modules:

- **lookit-api**: The repo containing the Experimenter Django app. The Lookit Django app is also in this repo.
- **ember-lookit-frameplayer**: A small Ember app that allows the API in lookit-api to talk to the exp-player and provides the rendering engine and experiment frames for Lookit studies

Generally, all ‘frame’ development will happen in ember-lookit-frameplayer.

To start developing your own frames, you will want to first follow the “Setup for local frame development” steps. To use the frame definitions you have created when posting a study on Lookit, you can specify your own ember-lookit-frameplayer repo to use (see “Using the experimenter interface”).



## 2.2.2 Getting Started

One of the features of **Ember CLI** is the ability to provide ‘blueprints’ for code. These are basically just templates of all of the basic boilerplate needed to create a certain piece of code. To begin developing your own frame:

```
cd ember-lookit-frameplayer/lib/exp-player
ember generate exp-frame exp-<your_name>
```

Where `<your_name>` corresponds with the frame name of your choice.

### 2.2.2.1 A Simple Example

Let’s walk through a basic example of ‘exp-consent-form’:

```
$ ember generate exp-frame
installing exp-frame
  create addon/components/exp-consent-form/component.js
  create addon/components/exp-consent-form/template.hbs
  create app/components/exp-consent-form.js
```

Notice this created three new files: - `addon/components/exp-consent-form/component.js`: the JS file for your ‘frame’ - `addon/components/exp-consent-form/template.hbs`: the Handlebars template for your ‘frame’ - `app/components/exp-consent-form.js`: a boilerplate file that exposes the new frame to the Ember app- you will almost never need to modify this file.

Let’s take a deeper look at the `component.js` file:

```
import ExpFrameBaseComponent from 'exp-player/components/exp-frame-base/component';
import layout from './template';

export default ExpFrameBaseComponent.extend({
  type: 'exp-consent-form',
  layout: layout,
  meta: {
    name: 'ExpConsentForm',
    description: 'TODO: a description of this frame goes here.',
    parameters: {
      type: 'object',
      properties: {
        // define configurable parameters here
      }
    },
    data: {
      type: 'object',
      properties: {
        // define data to be sent to the server here
      }
    }
  }
});
```

The first section:

```
import ExpFrameBaseComponent from 'exp-player/components/exp-frame-base';
import layout from './template';

export default ExpFrameBaseComponent.extend({
```

(continues on next page)

(continued from previous page)

```
    type: 'exp-consent-form',
    layout: layout,
    ...
  })
```

does several things: - imports the `ExpFrameBaseComponent`: this is the superclass that all ‘frames’ must extend  
- imports the `layout`: this tells Ember what template to use - extends `ExpFrameBaseComponent` and specifies `layout: layout`

Next is the ‘meta’ section:

```
    ...
    meta: {
      name: 'ExpConsentForm',
      description: 'TODO: a description of this frame goes here.',
      parameters: {
        type: 'object',
        properties: {
          // define configurable parameters here
        }
      },
      data: {
        type: 'object',
        properties: {
          // define data to be sent to the server here
        }
      }
    }
    ...
```

which is composed of: - name (optional): A human readable name for this ‘frame’ - description (optional): A human readable description for this ‘frame’. - parameters: JSON Schema defining what configuration parameters this ‘frame’ accepts. When you define an experiment that uses the frame, you will be able to specify configuration as part of the experiment definition. Any parameters in this section will be automatically added as properties of the component, and directly accessible as `propertyName` from templates or component logic. - data: JSON Schema defining what data this ‘frame’ outputs. Properties defined in this section represent properties of the component that will get serialized and sent to the server as part of the payload for this experiment. You can get these values by binding a value to an input box, for example, or you can define a custom computed property by that name to have more control over how a value is sent to the server.

If you want to save the value of a configuration variables, you can reference it in both parameters *and* data. For example, this can be useful if your experiment randomly chooses some frame behavior when it loads for the user, and you want to save and track what value was chosen.

It is important that any fields you define in `data` be named in camelCase: they can be all lowercase or they can be writtenLikeThis, but they should not start with capital letters or include underscores. This is because the fields from the Ember app will be converted to snake\_case for storage in the Postgres database, and may be converted back if another frame in Ember uses values from past sessions. We are fine if we go `fieldName` -> `field_name` -> `fieldName`, but anything else gets dicey! (Note to future developers: some conversations about this decision are available if this becomes a point of concern.)

### 2.2.2.2 Building out the Example

Let’s add some basic functionality to this ‘frame’. First define some of the expected parameters:

```

...
  meta: {
    ...,
    parameters: {
      type: 'object',
      properties: {
        title: {
          type: 'string',
          default: 'Notice of Consent'
        },
        body: {
          type: 'string',
          default: 'Do you consent to participate in this study?'
        },
        consentLabel: {
          type: 'string',
          default: 'I agree'
        }
      }
    }
  },
},
...

```

And also the output data:

```

...,
  data: {
    type: 'object',
    properties: {
      consentGranted: {
        type: 'boolean',
        default: false
      }
    }
  }
},
...

```

Since we indicated above that this ‘frame’ has a `consentGranted` property, let’s add it to the ‘frame’ definition:

```

export default ExpFrameBaseComponent.extend({
  ...,
  consentGranted: null,
  meta: {
    ...
  }
},
...

```

Next let’s update `template.hbs` to look more like a consent form:

```

<div class="well">
  <h1>{{ title }}</h1>
  <hr>
  <p>{{ body }}</p>
  <hr>
  <div class="input-group">
    <span>

```

(continues on next page)

(continued from previous page)

```

        {{ consentLabel }}
      </span>
      {{input type="checkbox" checked=consentGranted}}
    </div>
  </div>
<div class="row exp-controls">
  <!-- Next/Last/Previous controls. Modify as appropriate -->
  <div class="btn-group">
    <button class="btn btn-default" {{ action 'previous' }} > Previous </button>
    <button class="btn btn-default pull-right" {{ action 'next' }} > Next </button>
  </div>
</div>

```

We don't want to let the participant navigate backwards or to continue unless they've checked the box, so let's change the footer to:

```

<div class="row exp-controls">
  <div class="btn-group">
    <button class="btn btn-default pull-right" disabled={{ consentNotGranted }} {{
    ↪action 'next' }} > Next </button>
  </div>
</div>

```

Notice the new property `consentNotGranted`; this will require a new computed field in our JS file:

```

meta: {
  ...
},
consentNotGranted: Ember.computed.not('consentGranted')
});

```

### 2.2.3 Adding CSS styling

You will probably want to add custom styles to your frame, in order to control the size, placement, and color of elements. Experimenters use a common web standard called [CSS](#) for styles.\*

To add custom styles for a pre-existing component, you will need to create a file `<component-name>.scss` in the `styles/components` directory of `ember-lookit-frameplayer`. Then add a line to the top of `styles/app.scss`, telling it to use that style. For example,

```
@import "components/exp-video-physics";
```

Remember that anything in `ember-lookit-frameplayer` is shared code. Below are a few good tips to help your new frame stay isolated and distinct, so that it does not affect other projects.

- To protect other frames from being affected by your new styles, add a class of the same name as your frame (e.g., `exp-myframe`) to the div enclosing your component. Then prefix *every* rule in your `.scss` file with `.exp-myframe` to ensure that only your own frame is affected. Until we have a better solution, this practice will be enforced if you submit a pull request to add your frames to the common Lookit `ember-lookit-frameplayer` repo.
- To help protect your *own* frame's styling from possible future style changes (improperly) added by other people, you can give new classes and IDs in your component a unique prefix, so that they don't inadvertently overlap with styles for other things. For example, instead of `video-widget` and `should-be-centered`, use names like `exp-myframe-video-widget` and `exp-myframe-should-be-centered`.

Researchers using your frame can force it to be shown fullscreen (even if that is not the typical intended use) by passing the parameter `displayFullscreenOverride`. If you have not also set the `displayFullscreen` property of your frame to `true`, then the `#experiment-player` element will have class `player-fullscreen-override` but not `player-fullscreen`, to allow display to more closely mimic what it would be in non-fullscreen mode for things like forms and text pages.

If you create an (intentionally) fullscreen frame, then the element you make fullscreen will have class `player-fullscreen` while it is fullscreen, which you can use for styling.

\* You may notice that style files have a special extension `.scss`. That is because styles in experimenter are actually written in [SASS](#). You can still write normal CSS just fine, but SASS provides additional syntax on top of that and can be helpful for power users who want complex things (like variables).

## 2.2.4 Using mixins

Sometimes, you will wish to add a preset bundle of functionality to any arbitrary experiment frame. The Experimenter platform provides support for this via *mixins*.

To use a mixin for video recording, fullscreen, etc., simply have your frame “extend” the mixin. For instance, to use the `VideoRecord` mixin, your `component.js` file would define:

```
import ExpFrameBaseComponent from 'exp-player/components/exp-frame-base/component';
import layout from './template';

export default ExpFrameBaseComponent.extend(VideoRecord, {
  type: 'exp-consent-form',
  layout: layout,
  meta: {
    ...
  }
});
```

Your frame can extend any number of mixins. For now, be careful to check, when you use a mixin, that your frame does not defining any properties or functions that will conflict with the mixin’s properties or functions. If the mixin has a function `doFoo`, you can use that from your frame simply by calling `this.doFoo()`.

Below is a brief introduction to each of the common mixins; for more detail, see sample usages throughout the `ember-lookit-frameplayer` codebase and the mixin-specific docs [here](#)

### 2.2.4.1 FullScreen

This mixin is helpful when you want to show something (like a video) in fullscreen mode without distractions. You will need to specify the part of the page that will become full screen. By design, most browsers require that you interact with the page to trigger fullscreen mode.

### 2.2.4.2 MediaReload

If your component uses video or audio, you will probably want to use this mixin. It is very helpful if you ever expect to show two consecutive frames of the same type (eg two physics videos, or two things that play an audio clip). It automatically addresses a quirk of how ember renders the page; see [stackoverflow post](#) for more information.

### 2.2.4.3 VideoRecord

Functionality related to video capture, in conjunction with the [Pipe](#) system, for which MIT has a license.

## 2.2.5 Documenting your frame

We use [YUIDoc](#) for generating “automatic” documentation of ember-lookit-frameplayer frames, available [here](#). If you want to contribute your frames to the main Lookit codebase, please include YUIDoc-formatted comments following the example of existing frames, e.g. `exp-lookit-exit-survey`. Make sure to include:

- A general description of your frame
- An example of using it (the relevant JSON for a study)
- All inputs
- All outputs (data saved)
- Any events recorded

To check how your documentation will appear, run `yarn run docs` from the `ember-lookit-frameplayer` directory, then use `yuidoc --server` to see the docs served locally.

Include a screenshot in your frame documentation if possible! If your frame kind is `exp-smithlab-monkey-game`, name the screenshot `exp-player/screenshots/ExpSmithlabMonkeyGame.png` (i.e., go from dashes to CamelCase). For a simple frame, an actual screenshot is fine. If there are several “phases” to your frame or different ways it can work, you may want to make a diagram instead. When you run `yarn run docs`, this screenshot gets copied over to the YUIDoc theme for the project and to the `docs/assets` directory. The former is used locally, the latter when serving from github pages. Both the copy in `exp-player/screenshots` and the one in `docs/assets` should be committed using git; the one in the theme directory doesn’t have to be.

## 2.2.6 Ember debugging

Values of variables used in your frame are tricky to access directly from the Javascript console in your browser during testing.

There’s an [Ember Inspector browser plugin](#) you can use to help debug the Lookit components. Once you’ve installed it, you’ll find it along with other developer tools.

Here’s how to find relevant data for a particular frame. Screenshots below are for Google Chrome.

This lets you right away change any of the data you sent to the frame in the JSON document. E.g., on the consent page, try changing the “prompt” to something else. If something is going wrong, hopefully this information will be helpful.

You can send the entire component (or anything else) to the console using the little `>$E` button:

And then to keep using it, save it as a variable:

Then you can do things like try out actions, e.g. `this.send`.

## 2.2.7 When should I use actions vs functions?

Actions should be used when you need to trigger a specific piece of functionality via user interaction: eg click a button to make something happen.

Functions (or helper methods on a component/frame) should be used when the logic is shared, or not intended to be accessed directly via user interaction. It is usually most convenient for these methods to be defined as a part of the component, so that they can access data or properties of the component. Since functions can return a value, they are particularly helpful for things like sending data to a server, where you need to act on success or failure in order to display information to the user. (using promises, etc)

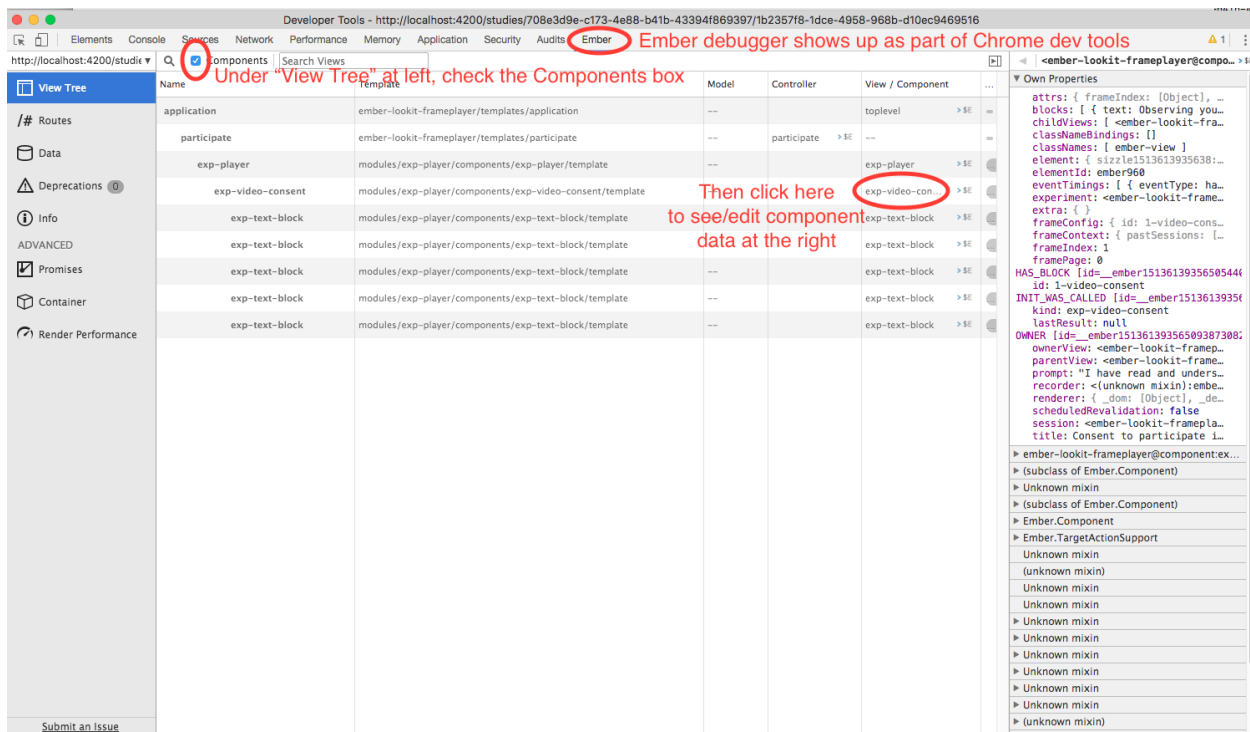


Fig. 1: Ember debugger tree view

Usually, you should use actions only for things that the user directly triggers. Actions and functions are not mutually exclusive! For example, an action called `save` might call an internal method called `this._save` to handle the behavior and message display consistently.

If you find yourself using the same logic over and over, and it does not depend on properties of a particular component, consider making it a `util`!

If you are building extremely complex nested components, you may also benefit from reading about closure actions. They can provide a way to act on success or failure of something, and are useful for : - [Ember closure actions have return values](#) - [Ember.js Closure Actions Improve the Former Action Infrastructure](#)

## 2.3 How to capture video in your frame

Webcam video recording during Lookit frames is currently accomplished using WebRTC as the interface to the webcam and [Pipe](#) for video streaming and processing.

Lookit frames that collect video data make use of an Ember mixin `VideoRecord` included in `ember-lookit-frameplayer`, which makes a `VideoRecorderObject` available for use in the code for that frame. This object includes methods for showing/hiding the webcam view, starting/pausing/resuming/stopping video recording, installing/destroying the recorder, and checking the current video timestamp (see <https://lookit.github.io/ember-lookit-frameplayer/classes/VideoRecorderObject.html>). The programmer designing a new frame can therefore flexibly indicate when recording should begin and end, as well as recording video timestamps for any events recorded during this frame (e.g., so that during later data analysis, researchers know the exact time in the video where a new stimulus was presented). The name(s) of any videos collected during a particular frame as included in the session data recorded, to facilitate matching sessions to videos; video filenames also include the study ID, session ID, frame ID, and a timestamp.

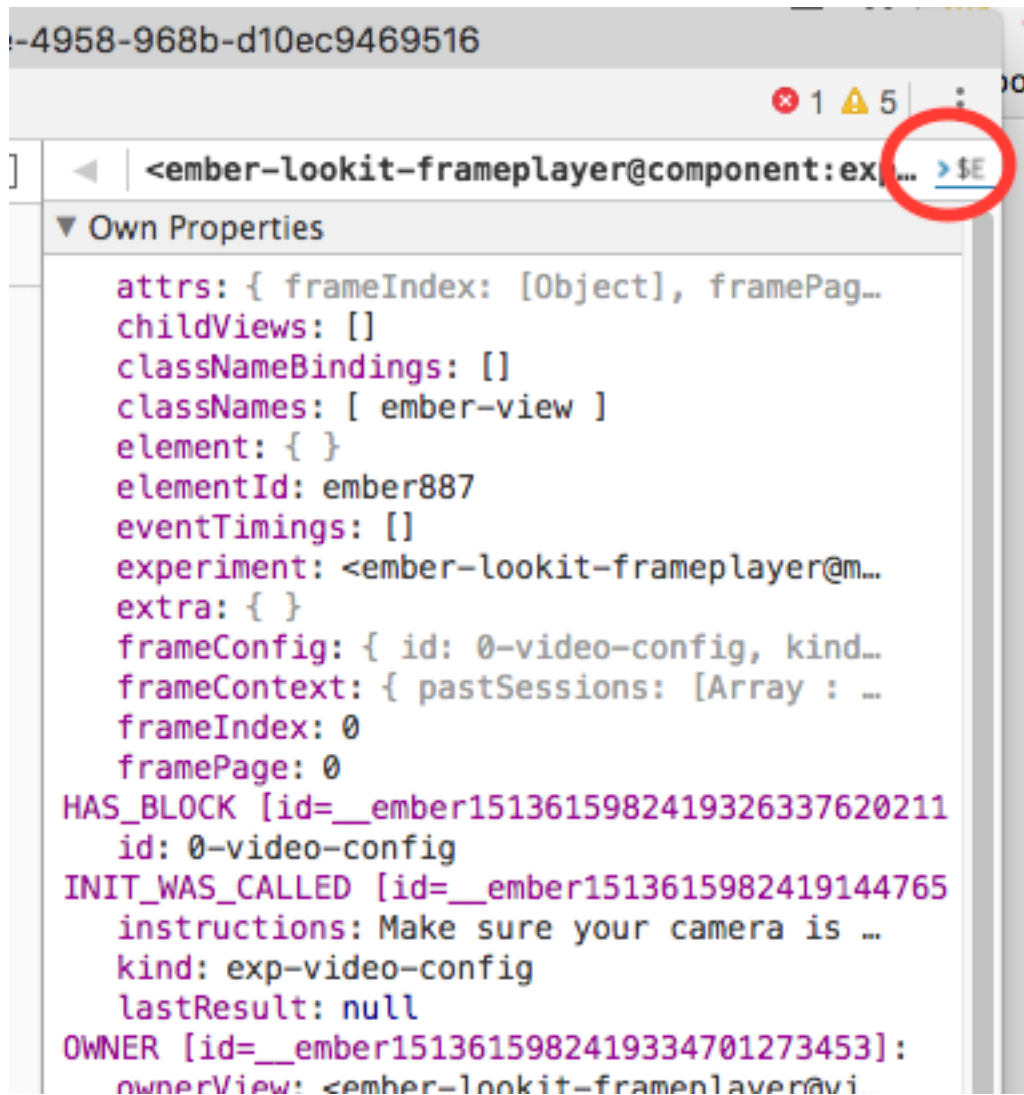


Fig. 2: Ember debugger send to console

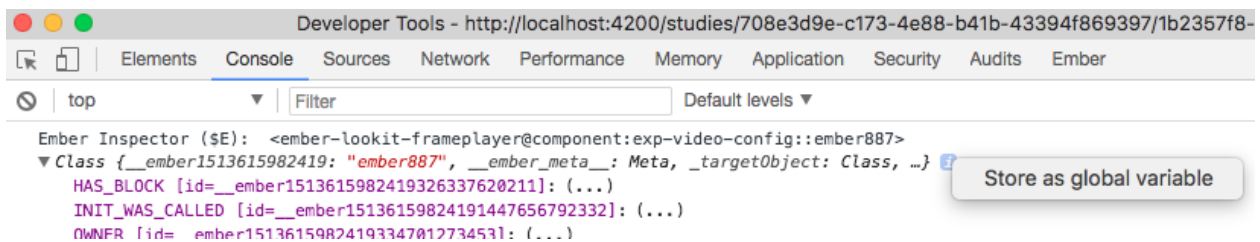


Fig. 3: Ember debugger save variable



To begin, you will want to add the `VideoRecord` mixin to your experiment frame. This provides, but does not in itself activate, the capability for your frame to record videos.

```
import ExpFrameBaseComponent from '../../components/exp-frame-base/component';
import VideoRecord from '../../mixins/video-record';

export default ExpFrameBaseComponent.extend(VideoRecord, {
  // Your code here
});
```

### 2.3.1 Limitations

One technical challenge imposed by webcam video streaming is that a connection to the server must be established before webcam recording can be quickly turned on and off, and this process may take up to several seconds. Each experiment frame records a separate video clip and establishes a separate connection to the server, so frames must be designed to wait for recording to begin before proceeding to a portion of the trial where video data is required. This fits well with typical study designs using looking time or preferential looking, where the child's attention is returned to the center of the screen between trials; the first few seconds of the child watching the "attention grabber" are not critical and we can simply ensure that the webcam connection is established before proceeding to the actual experimental trial. When collecting verbal responses, the study frame can simply pause until the connection is established or, similarly, proceed with an initial portion of the trial where video data is not required.

Currently, continuous webcam recording across frames is not possible on Lookit; any periods of continuous recording must be within a single frame. This is not a hard technical limitation, though.

### 2.3.2 How it works

The `VideoRecord` mixin is how a new frame makes use of video recording functionality. In turn, this mixin uses the video-recorder service, which relies on [Pipe](#). To set everything up from scratch, e.g. if you're creating Mookit, an online experimental platform for cows, you'll need to do the following:

- Make a Pipe account, and get the account hash and environment ID where you want to send videos.
- Create an Amazon S3 bucket (where video will be sent by Pipe, then renamed). Set up Pipe to send your videos to this bucket; you'll need to create an access key that just allows putting videos in this bucket. Go to IAM credentials, and make a group with the following policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "s3:ListAllMyBuckets",
      "Resource": "*"
    },
    {
      "Sid": "VisualEditor1",
      "Effect": "Allow",
      "Action": "s3:GetBucketLocation",
      "Resource": "arn:aws:s3:::*"
    },
    {
      "Sid": "VisualEditor2",
      "Effect": "Allow",
```

(continues on next page)

(continued from previous page)

```

        "Action": "s3:PutObject",
        "Resource": [
            "arn:aws:s3:::MYBUCKET/*",
            "arn:aws:s3:::MYBUCKET"
        ]
    }
]
}

```

Then make a user, and add it to your new group. Use the keys for this user in Pipe.

- Create a webhook key in Pipe, and store it in as `PIPE_WEBHOOK_KEY` in the lookit-api Django app `.env` file. This will let Lookit rename the video files to something sensible upon being uploaded to S3.
- Create a webhook in Pipe for the event `video_copied_s3`, and send it to `https://YOURAPP/exp/renamevideo/`
- Store the `PIPE_ACCOUNT_HASH` and `PIPE_ENVIRONMENT` in the ember-lookit-frameplayer `.env` file. This is what lets Lookit video go to the right Pipe account.

## 2.4 Custom randomizer frames

Experimenter supports a special kind of frame called ‘choice’ that defers determining what sequence of frames a participant will see until the page loads. This allows for dynamic ordering of frame sequence in particular to support randomization of experimental conditions or counterbalancing. The goal of this page is to walk through an example of implementing a custom ‘randomizer’.

### 2.4.1 Overview of ‘choice’ structure

Generally the structure for a ‘choice’ type frame takes the form:

```

{
  "kind": "choice",
  "sampler": "random",
  "options": [
    "video1",
    "video2"
  ]
}

```

Where: - **sampler** indicates which ‘randomizer’ to use. This must correspond with the values defined in `lib/exp-player/addon/randomizers/index.js` - **options**: an array of options to sample from. These should correspond with values from the `frames` object defined in the experiment structure (for more on this, see [the experiments docs](#))

### 2.4.2 Making your own

There is some template code included to help you get started. From within the `ember-lookit-frameplayer/lib/exp-player` directory, run:

```
ember generate randomizer <name>
```

which will create a new file: `addon/randomizers/<name>.js`. Let's walk through an example called 'next'. The 'next' randomizer simply picks the next frame in a series. (based on previous times that someone participated in an experiment)

```
$ ember generate randomizer next
...
installing randomizer
  create addon/randomizers/next.js
```

Which looks like:

```
/*
  NOTE: you will need to manually add an entry for this file in addon/randomizers/
  ↪index.js, e.g.:
import
import Next from './next';
...
{
  ...
  next: Next
}
*/
var randomizer = function(/*frame, pastSessions, resolveFrame*/) {
  // return [resolvedFrames, conditions]
};
export default randomizer;
```

The most important thing to note is that this module exports a single function. This function takes three arguments: - `frame`: the JSON entry for the 'choice' frame in context - `pastSessions`: an array of this participants past sessions of taking this experiment. See [the experiments docs](#) for more explanation of this data structure - `resolveFrame`: a copy of the ExperimentParser's `_resolveFrame` method with the `this` context of the related ExperimentParser bound into the function.

Additionally, this function should return a two-item array containing: - a list of resolved frames - the conditions used to determine that resolved list

Let's walk through the implementation:

```
var randomizer = function(frame, pastSessions, resolveFrame) {
  pastSessions = pastSessions.filter(function(session) {
    return session.get('conditions');
  });
  pastSessions.sort(function(a, b) {
    return a.get('createdOn') > b.get('createdOn') ? -1: 1;
  });
  // ...etc
};
```

First we make sure to filter the `pastSessions` to only the one with reported conditions, and make sure the sessions are sorted from most recent to least recent.

```
...
var option = null;
if(pastSessions.length) {
  var lastChoice = (pastSessions[0].get(`conditions.${frame.id}`) || frame.
  ↪options[0]);
  var offset = frame.options.indexOf(lastChoice) + 1;
  option = frame.options.concat(frame.options).slice(offset)[0];
}
```

(continues on next page)

(continued from previous page)

```
}  
else {  
  option = frame.options[0];  
}
```

Next we look at the conditions for this frame from the last session (`pastSessions[0].get(conditions.${frame.id})`). If that value is unspecified, we fall back to the first option in `frame.options`. We calculate the index of that item in the available `frame.options`, and increment that index by one.

This example allows the conditions to “wrap around”, such that the “next” option after the last one in the series circles back to the first. To handle this we append the `options` array to itself, and slice into the resulting array to grab the “next” item.

If there are not past sessions, then we just grab the first item from `options`.

```
var [frames,] = resolveFrame(option);  
return [frames, option];  
};  
  
export default randomizer;
```

Finally, we need to resolved the selected sequence using the `resolveFrame` argument. This function always returns a two-item array containing: - an array of resolved frames - the conditions used to generate that array

In this case we can ignore the second part of the return value, and only care about the returned `frames` array.

The `export default randomizer` tells the module importer that this file exports a single item (`export default`), which in this case is the `randomizer` function (**note**: the name of this function is not important).

Finally, lets make sure to add an entry to the `index.js` file in the same directory:

```
import next from './next';  
  
export default {  
  ...,  
  next: next  
};
```

This allows consuming code to easily import all of the randomizers at once and to index into the `randomizers` object dynamically, e.g. (from the `ExperimentParser`):

```
import randomizers from 'exp-player/randomizers/index';  
// ...  
return randomizers[randomizer] (  
  frame,  
  this.pastSessions,  
  this._resolveFrame.bind(this)  
);
```

---

## Installation: lookout-api (Django project)

---

`lookout-api` is the codebase for Experimenter and Lookit, excluding the actual studies themselves. Any functionality you see as a researcher or a participant (e.g., signing up, adding a child, editing or deploying a study, downloading data) is part of the `lookout-api` repo. The Experimenter platform is the part of this project for designing and administering research studies, meant for researchers. The Lookit platform is participant-facing, where users can signup and take part in studies. This project is built using Django and PostgreSQL. (The studies themselves use Ember.js; see Ember portion of codebase, [ember-lookout-frameplayer](#).), It was initially developed by the [Center for Open Science](#).

If you install only the `lookout-api` project locally, you will be able to edit any functionality that does not require actual study participation. For instance, you could contribute an improvement to how studies are displayed to participants or create a new CSV format for downloading data as a researcher.

Note: These instructions are for Mac OS. Installing on another OS? Please consider documenting the exact steps you take and submitting a PR to the `lookout-api` repo to update the documentation!

### 3.1 Prerequisites

- Make sure you have python 3.6: `$ python --version` will check the version of your current default python installation. If you don't have this, install from <https://www.python.org/>.
- Make sure you have pip. `$ pip --version`
- Create a virtual environment using python 3.6
  - One way to do this:
  - `$ pip install virtualenv`
  - `$ virtualenv -p python3 envname`, where “*envname*” is the name of your virtual environment.
  - `$ source envname/bin/activate` Activates your virtual environment
- Install postgres
  - make sure you have brew `$ brew`

- `$ brew install postgresql`
- `$ brew services start postgres` *Starts up postgres*
- `$ createdb lookit` *Creates lookit database*

## 3.2 Installation

- `$ git clone https://github.com/lookit/lookit-api.git`
- `$ cd lookit-api`
- `$ sh up.sh` *Installs dependencies and run migrations*
- `$ python manage.py createsuperuser` *Creates superuser locally (has all user permissions)*
- `$ touch project/settings/local.py` *Create a local settings file.*
- Add `DEBUG = True` to `local.py` and save. This is for local development only.
- `$ python manage.py runserver` *Starts up server*

## 3.3 Authentication

OAuth authentication to OSF accounts, used for access to Experimenter, currently does not work when running locally. You can create a local participant account and log in using that to view participant-facing functionality, or log in as your superuser at `localhost:8000/admin` and then navigate to Experimenter. As your superuser, you can also use the Admin app to edit other local users - e.g., to make users researchers vs participants, in particular organizations, etc.

## 3.4 Handling video

This project includes an incoming webhook handler for an event generated by the Pipe video recording service when video is transferred to our S3 storage. This requires a webhook key for authentication. It can be generated via our Pipe account and, for local testing, stored in `project/settings/local.py` as `PIPE_WEBHOOK_KEY`. However, Pipe will continue to use the handler on the production/staging site unless you edit the settings to send it somewhere else (e.g., using ngrok to send to localhost for testing).

## 3.5 Common Issues

During the installation phase, when running `sh up.sh`, you may see the following:

```
psql: FATAL:  role "postgres" does not exist
```

To fix, run something like the following from your home directory:

```
$ ../../../../usr/local/Cellar/postgresql/9.6.3/bin/createuser -s postgres
```

If your version of postgres is different than 9.6.3, replace with the correct version above. Running this command should be a one-time thing.

You might also have issues with the installation of `pygraphviz`, with errors like

```
running install
Trying pkg-config
Package libcgraph was not found in the pkg-config search path.
Perhaps you should add the directory containing `libcgraph.pc'
to the PKG_CONFIG_PATH environment variable
No package 'libcgraph' found
```

or

```
pygraphviz/graphviz_wrap.c:2954:10: fatal error: 'graphviz/cgraph.h' file not found
#include "graphviz/cgraph.h"
      ^
1 error generated.
error: command 'clang' failed with exit status 1
```

To fix, try running something like:

```
$ brew install graphviz
$ pip install --install-option="--include-path=/usr/local/include" --install-option="--library-path=/usr/local/lib" pygraphviz
```

Then run `sh up.sh` again.

## 3.6 Continued Installation for developers

If you want to work on the functionality of Lookit, there are a few more installations steps you need to take. More information about the following programs can be found in the *Technical Glossary*

### 3.6.1 Install Docker

Follow [these instructions](#). Make sure you select the proper OS.

### 3.6.2 Install Postgres

#### For Linux Users

Before you get started, update your system with this command:

```
sudo apt-get update
```

Make sure you have python3 and pip installed:

```
sudo apt install python3
```

```
sudo apt install python-pip
```

Now, begin to install Postgres:

```
sudo apt-get install PostgreSQL PostgreSQL-contrib
```

Run the following command. It will take inside the Postgres world.

```
sudo -u postgres psql postgres
```

Every command now should start with `postgres=#`

In the postgres world, run the following commands:

```
#\password postgres
```

You should be prompted to enter a new password. Don't type anything, just hit enter twice. This should clear the password.

```
postgres=# create database lookit
```

```
postgres=# grant all privileges on database lookit to postgres
```

If at this point you still do not have access to the lookit database, run the following commands:

```
sudo vi /etc/postgresql/10/main/pg_hba.conf
```

```
pg_ctl reload
```

A long document with # on the leftmost side of almost every line should open up. Scroll to the bottom. There will be a few lines that don't start with #. They might be a different color and will start with either local or host. The last word in each of those lines should be trust. If it is not, switch into editing mode ( hit esc then type i and hit enter ) and change them to say trust. Then, save the file ( hit esc and then type :x before hitting enter ). You should now have access.

### 3.6.3 Install RabbitMQ

#### For Linux Users

First, run the following command:

```
sudo apt install rabbitmq-server
```

Now that rabbitmq server is installed, create an admin user with the following commands:

```
sudo rabbitmqctl add_user admin password
```

```
sudo rabbitmqctl set_user_tags admin administrator
```

```
sudo rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

Make sure that the server is up and running:

```
sudo systemctl stop rabbitmq-server.service
```

```
sudo systemctl start rabbitmq-server.service
```

```
sudo systemctl enable rabbitmq-server.service
```

If you are having problems creating a user or getting the server running, try the following commands:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

```
sudo chown -R rabbitmq:rabbitmq /var/lib/rabbitmq/
```

```
sudo rabbitmqadmin declare queue --vhost=/ name=email
```

```
sudo rabbitmqadmin declare queue --vhost=/ name=builds
```

```
sudo rabbitmqadmin list queues
```

When you run the last command, you should see the following ASCII art:

```
+-----+-----+
| name  | messages |
+-----+-----+
| builds|      0    |
| email |      0    |
+-----+-----+
```



### 3.6.4 Install Ngrok

#### For Linux Users

To install, run this command:

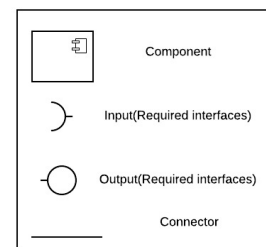
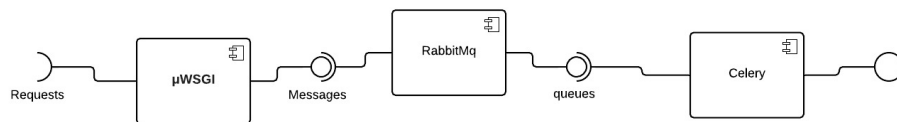
```
sudo snap install ngrok
```

To connect to your local host run this command:

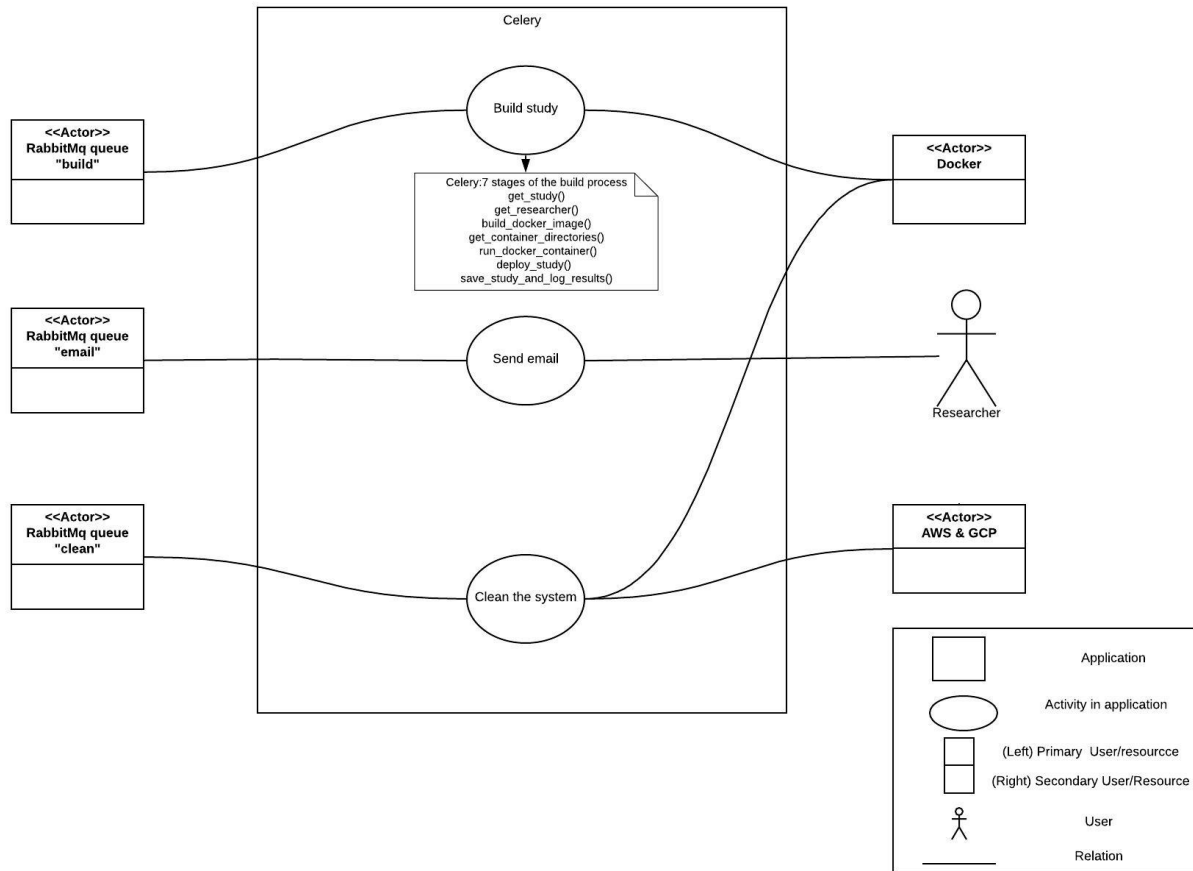
```
ngrok http "[https://localhost:8000] (https://localhost:8000) "
```

## 3.7 How Do These Programs Work Together?

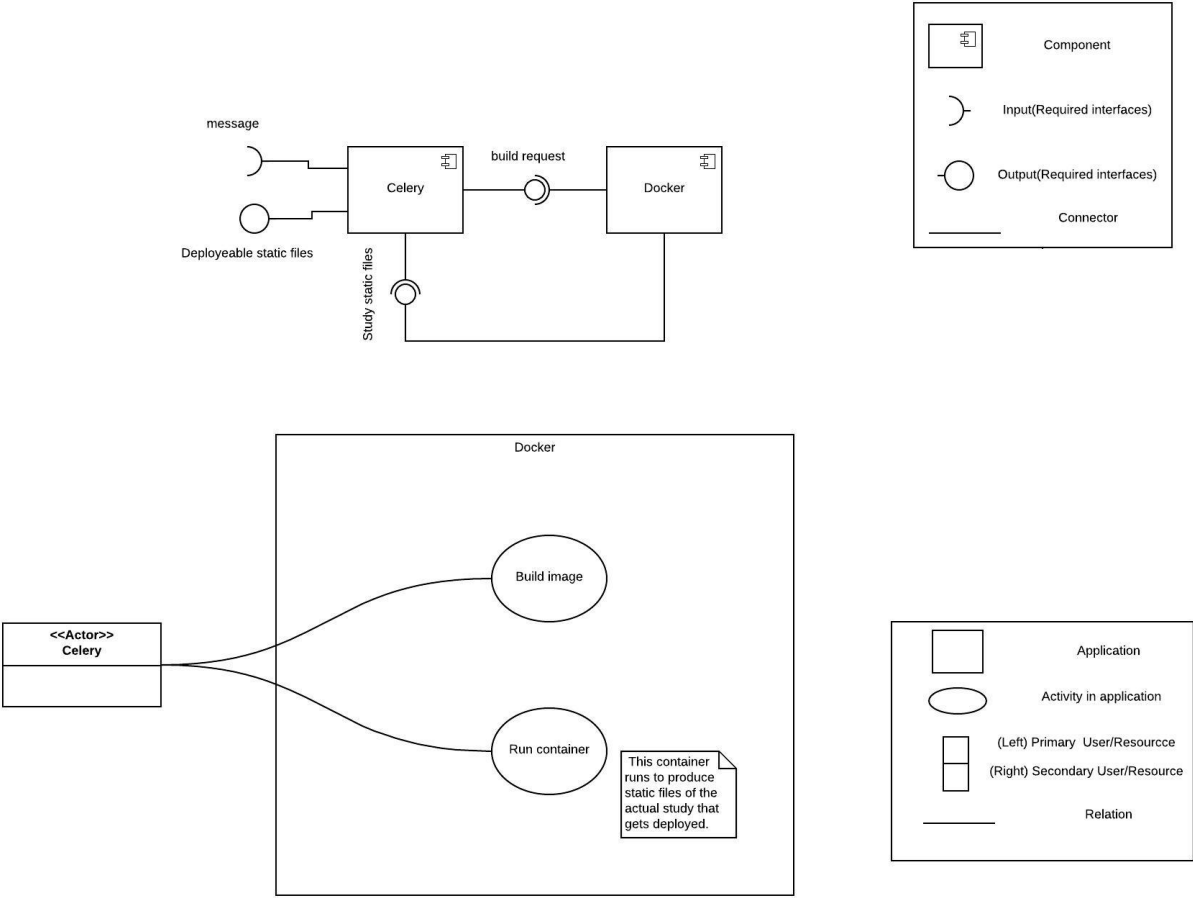
The following diagrams illustrate how different parts of the API interact with each other. For more information about these programs, please reference the *Technical Glossary*



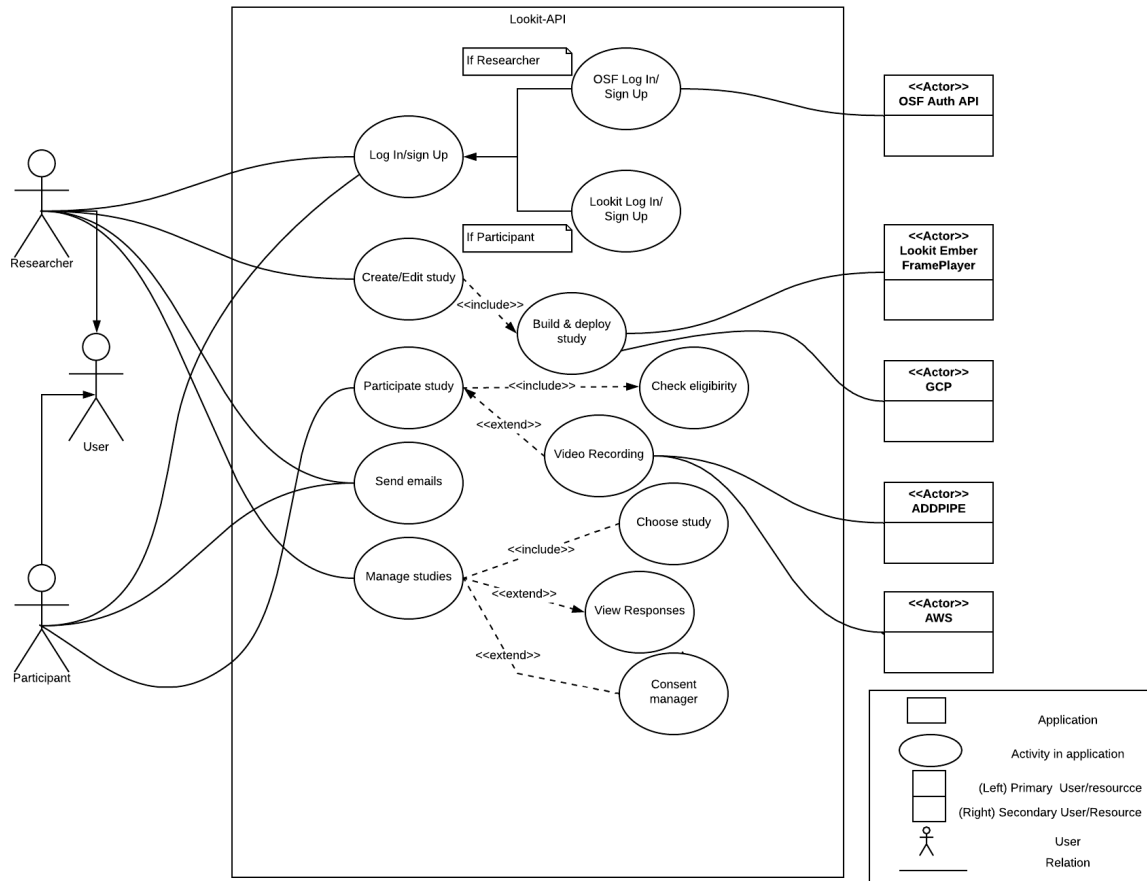
Every time the user makes a request, the request is sent through  $\mu$ WSGI. For certain requests that take more than a few seconds (e.g., building study dependencies, sending mail, etc.), the WSGI handler defers the work by kicking off a new celery task. The payload for this task is mediated by the RabbitMQ service. If the request is short enough, HTTP will handle the request on its own.



Celery is used to build and relay tasks and make Lookit more efficient. Lengthy requests are mediated by RabbitMQ for celery to complete on the side. The tasks sent to celery are ones that would ruin the user experience if they backlogged the HTTPs request cycle. Programs like celery are used to keep the request cycle short.



When you want to build a study, celery sends that request to Docker, which groups individual studies into nice containers for reproducibility of the experiment functionality and protection from later changes to the code. Docker then sends the study static files back to celery. After building the study, celery sends deployable static files to Google Cloud.



This is a diagram of all interactions possible with the Lookit API. On the rightmost side are all external resources being used/

---

### Installation: ember-lookit-frameplayer (Ember app)

---

`ember-lookit-frameplayer` is a small Ember application that allows both researchers to preview an experiment and users to participate in an experiment. This is meant to be used in conjunction with the [Lookit API Django project](#), which contains the Experimenter and Lookit applications. The Django application will proxy to these Ember routes for previewing/participating in an experiment.

In order to run the frame player as it works on Lookit, you will need to additionally install the Django app `lookit-api` and then follow the local frame development instructions to make sure it communicates with the Ember app. This way, for instance, an experiment frame will be able to look up previous sessions a user has completed and use those for longitudinal designs.

Note: These instructions are for Mac OS. Installing on another OS? Please consider documenting the exact steps you take and submitting a PR to the `lookit-api` repo to update the documentation!

## 4.1 Prerequisites

You will need the following tools properly installed on your computer.

- [Git](#)
- [Node.js](#) (with NPM)
- [Bower](#)

## 4.2 Installation

Before beginning, you will need to install Yarn, a package manager (like npm).

```
git clone https://github.com/lookit/ember-lookit-frameplayer.git
cd ember-lookit-frameplayer
yarn install --pure-lockfile
bower install
```

Create or open a file named `.env` in the root of the `ember-lookit-frameplayer` directory, and add the following entries to use the Pipe WebRTC-based recorder: `PIPE_ACCOUNT_HASH` (reference to account to send video to) and `PIPE_ENVIRONMENT` (which environment, e.g. development, staging, or production). These are available upon request if you need to use the actual Lookit environments. (If you are doing a very large amount of local testing, we may ask that you set up your own Pipe account.) Your `.env` file should look like this:

```
PIPE_ACCOUNT_HASH='<account hash here>'
PIPE_ENVIRONMENT=<environment here>
```

## 4.3 Running / Development

- `ember serve`
- Visit your app at <http://localhost:4200>.

If you change any dependencies, make sure to update and commit the `yarn.lock` file in addition to `package.json`.

### 4.3.1 Code Generators

Make use of the many generators for code, try `ember help generate` for more details

### 4.3.2 Running Tests

- `ember test`
- `ember test --server`

### 4.3.3 Building

- `ember build (development)`
- `ember build --environment production (production)`

### 4.3.4 Writing documentation of frames

Documentation of individual exp-player components is automatically generated using YUIDoc:

- `yarn run docs`

At the moment, this is a manual process: whatever files are in the top level `/docs/` folder of the master branch will be served via GitHub pages.

---

## Django app implementation notes

---

How various aspects of the Django app work (and why they were set up that way) that may be of interest to developers.

### 5.1 Permissions

#### 5.1.1 Generic best practices

- Groups are an important abstraction between users and permissions.
  - If you assign permissions directly to a user it will be difficult to find out who has the permissions and difficult to remove them.
- Creating a Group just to wrap an individual permission is fine.
- Include the model name when defining model specific permissions. Permissions are referenced with `app_name` and permission codename.
- Always check for individual permissions. **NEVER CHECK IF SOMEONE BELONGS TO A GROUP or “is\_superuser”**
- `is_superuser` implicitly grants all permissions to a user. Any permissions check will return `True` if a user `is_superuser`.

#### 5.1.2 Guardian, how does it work?

Django provides model-level permissions. That means that you can allow users in the Scientist group the ability to read the Report model or users in the Admin group the ability to create, read, update, and delete the Report model.

[Guardian](#) provides object-level permissions. That means that you can allow users in the Southern Scientists group the ability to read the a specific Report instance about Alabama.

Guardian does this by leveraging Django’s generic foreign key field. This means that Guardian can have a severe performance impact on queries where you check object-level permissions. It will cause a double join through Django’s ContentType table. If this becomes non-performant you can switch to using [direct foreign keys](#).

## 5.2 Workflow: managing study states

### 5.2.1 Why Transitions

`Transitions` is an object-oriented state machine implemented in Python.

It's both very powerful and very simple. It's definition is a python dictionary so it can be easily serialized into JSON and stored in a database or configured via YAML. It has callback functionality for state transitions. It can create diagrams of the workflow using pygraphviz. It also ties into django model classes very easily.

### 5.2.2 How

The workflow is defined in `studies/workflow.py` in a dictionary called `transitions`. Here is a [gist](#) that explains how the pieces fit together.

### 5.2.3 Make a diagram

To make a workflow diagram in png format start a shell plus instance with `python manage.py shell_plus` and execute the following:

```
# get a study you'd like to diagram
s = Study.objects.first()
# draw the whole graph ... in which case the study you choose doesn't matter
s.machine.get_graph().draw('fancy_workflow_diagram.png', prog='dot')
# ... or just the region of interest (contextual to the study you chose)
# (previous state, active state and all reachable states)
s.machine.get_graph(show_roi=True).draw('roi_diagram.png', prog='dot')
```

### 5.2.4 Logging

There is a `_finalize_state_change` method on the `Study` model. It fires after every workflow transition. It saves the model with its updated `state` field and also creates a `StudyLog` instance making record of the transition. This callback would be the optimal place to add functionality that needs to happen after every workflow transition.

## 5.3 Celery tasks

### 5.3.1 build\_experiment task

The business requirements for this project included the ability for experiments to rely on versioned dependencies without causing conflicts between experiments.

The experiment application is dependent on the `ember-lookit-frameplayer` repo. Researchers have the ability to specify a custom github url for the `ember-lookit-frameplayer` repo. They can also specify a SHA for the commit that they would like to use. These fields are on the Build Study Page.

### 5.3.2 What happens

The build process uses `celery`, `docker`, `ember-cli`, `yarn`, and `bower`.



When a build or preview is requested a celery task is put into the build queue.

Inside the task, the study and requesting user are looked up in the database. If it's a preview task it's current state is copied into a new variable to be saved for later, then the study is put into the state of `previewing` and saved. If it's a deployment the study is put into the state of `deployment` and saved. Since these states don't actually exist in the workflow definition this short circuits the workflow engine so that studies currently undergoing deployment or preview can neither move through the workflow or be previewed or deployed concurrently.

The SHAs are checked in the study model's metadata field, if they are empty or invalid the **HEAD** of the default branch is used. This requires HTTPS calls to github for the `ember-lookit-frameplayer` repository.

A zip file of each repo is downloaded to a temporary directory and extracted. The `lookit-frame-player` archive is extracted in the *checkout directory* (`ember_build/checkouts/{player_sha}`).

A docker image is built based on the `node:8.2.1-slim` image. It is rebuilt every time because it doesn't change very often and docker rebuilds of unchanged images are very fast.

The container is started passing several environment variables. It installs `python3` and several other dependencies with `apt-get`. Then it installs `yarn`, `bower`, and `ember-cli@2.8` globally with `npm`. Next it mounts the `ember-build/checkouts` directory to `/checkouts` inside the container and the `ember-build/deployments` directory to `/deployments` inside the container. It copies `ember-build/build.sh` and `ember-build/environment` into the root (`/`) of the container and executes `/bin/bash /build.sh`.

`build.sh` copies the contents of the *checkout directory* into the container (`/checkout-dir/` inside the container) for faster file access. A couple of `sed` replacements are done where there are experiment specific data that needs to be hardcoded prior to `ember-build` running. The `environment` files are copied into the correct places. Then `yarn install --pure-lockfile` and `bower install --allow-root` are run for `ember-lookit-frameplayer`. Once those have completed `ember-build -prod` is run to create a distributable copy of the app. The contents of the `dist` folder is then copied into the study output directory. The container is now destroyed.

Once the build process is finished the files in the `dist` folder are copied to a folder on Google Cloud Storage. If it's a preview they go into a `preview_experiments/{study.uuid}` folder in the bucket for the environment (staging or production). If it's a deployment they go into a `experiments/{study.uuid}` folder in the bucket for the environment (staging or production).

When the task is finished copying the files to Google Cloud Storage an email is sent to the study admins and organization admins.

If the task was a preview task the state of the study is set back to it's previous state. If it was a deployment the study is set to active. If the study is marked as discoverable, it will now be displayed on the lookit studies list page.

Finally, regardless of whether the task completed successfully a study log will be created. The extra field (a JSON field) will contain the logs of the image build process, the logs of the ember build process than ran inside the docker container, any raised exception, and the any logs generated by python during the entire task run. This is very helpful for debugging. Line endings are encoded for ease of storage so to read the results easily copy the contents of a study logs extra field from the admin into an editor and replace the overly escaped linebreaks (`\\\\\\n|\\n`) with actual line breaks `\n`. You can also use a JSON beautifier/formatter to aid readability.

### 5.3.3 build\_zipfile\_of\_videos

This task downloads videos from MIT's Amazon S3 bucket, zips them up, uploads them to Google Cloud Storage, generates a signed url good for 30m, and emails the requesting user that URL.

- The zip filename is generated from the study uuid, a sha256 of the included filenames, and whether it's consent videos or all videos.
- If a zip file exists on Google Cloud Storage with the same name the file is not regenerated, an email with a link is immediately sent.

- After the task is completed all video files are immediately removed from the server. They still exist on s3 and Google Cloud Storage.

### **5.3.4 cleanup\_builds**

This finds build directories older than a day and deletes them. It's scheduled to run every morning at 2am.

### **5.3.5 cleanup\_docker\_images**

This finds unused docker images from previous builds and deletes them. It's scheduled to run every morning at 3am.

### **5.3.6 cleanup\_checkouts**

This finds checkout (extracted archives of github repos) directories older than a day and deletes them. It's scheduled to run every morning at 4am.

---

## Guidelines for contributors

---

Interested in helping write the code behind the Lookit platform? Thanks for supporting open source science! This page describes the process any would-be contributor should plan to use. We have included some beginner-friendly details in case you are new to open source projects.

The content of this page applies to all three Lookit repos: `lookit-api` (Lookit site), `ember-lookit-frameplayer` (system for displaying experiments & components to use), and `lookit-docs` (specific frames, subrepo of `ember-lookit-frameplayer`).

>> **Where's the code I need?** If you only want to change something about the Lookit site, without touching experiment functionality (for instance, to add a question to the demographic survey or change how studies are sorted), you will only need to run `lookit-api` and can follow the Django project installation steps. If you want to develop experiment frames or change how the experiment player works, you will need to follow the steps for local frame development, installing *both* `lookit-api` and `ember-lookit-frameplayer` and telling them how to talk to each other. Your changes, however, will likely be limited to `ember-lookit-frameplayer`.

### 6.1 Prerequisites

To contribute to the `lookit-api` codebase, it will be very helpful to have (a) a strong grasp of Python and (b) some familiarity with the Django framework. Learning Python is outside the scope of these docs, but if you want someplace to start, we highly recommend [Think Python](#). If you're already familiar with Python but haven't used the web framework Django, we highly recommend taking the time to complete [the official tutorial](#).

To contribute to the `ember-lookit-frameplayer` codebase - e.g., when creating your own experiment frames - it will be helpful to have (a) a strong grasp of Javascript and (b) some familiarity with Ember.js. However, we're really not using that much of the functionality of Ember, and if you're just making some new frames, we would recommend getting started by trying out modifications of an existing frame to get your feet wet, rather than trying to learn Ember from scratch.

To contribute to these docs, you'll just need to be able to edit [ReStructured Text](#) files! You don't need to learn anything in advance - just look up syntax when you're not sure how to make a link, include an image, etc.

## 6.2 Getting started

At a high level, we are roughly following a Forking Workflow version of Gitflow [as described here](#). You should plan to make feature-specific branches off of the `develop` branch of a local copy of the code running on your own machine. This will keep the codebase as clean as possible. Before submitting a PR, merge in the most recent changes from the `develop` branch.

First create your own fork of `lookit-api`, `ember-lookit-frameplayer`, and/or `lookit-docs`. Follow the directions for installation of `lookit-api` or `ember-lookit-frameplayer` if needed.

## 6.3 Ignoring some files

You may want to configure a global `.gitignore` on your machine and include your `virtualenv(s)` along with any files specific to your system. A sample global `.gitignore` is available [here](#) – you can tell git to globally ignore files specified in a `.gitignore` file via:

```
git config --global core.excludesfile ~/path/to/your/.gitignore_global
```

## 6.4 Add your own feature and submit a Pull Request

Keep your commit history clean and merge process simple by following these steps before starting on any new feature.

One time only, add the original repo as a remote to your fork, e.g., if you are contributing to *lookit-api* you would run a command like this:

SSH:

```
git remote add upstream git@github.com:lookit/lookit-api.git
```

HTTPS:

```
git remote add upstream https://github.com/lookit/lookit-api.git
```

Anytime a PR is merged or changes are pushed (or you're starting this process for the first time), you should run:

```
git checkout develop
git pull upstream develop
```

in order to make sure you are working with an up-to-date copy of the *develop* branch.

Once you have the most recent *develop* code, pick an issue (or create a new one) which your new feature will address and create a new branch off of *develop*. Note: our project convention is to prepend *feature/* or *hotfix/* to the feature or issue name for a richer annotation of the commit history.

If you want to create a new validation feature, for example, you might name it like this:

```
git checkout -b feature/my-validation-feature
```

Now you can run *git branch* and should see an output like this:

```
$ git branch
  develop
  master
* feature/my-validation-feature
```

Proceed with writing code. Commit frequently! Focus on writing very clear, concise commit statements and plentiful comments. If you have poor comments or zero tests, your PR will not be merged.

If you are aware of changes in the branch you forked from, rebase your branch from that changing branch (in our case that is *develop*) by running:

```
git rebase develop
```

and then resolving all merge conflicts.

On *lookit-api*, you should then update dependencies like this:

```
pip install -r requirements/defaults.txt
python manage.py migrate
python manage.py test
```

On *ember-lookit-frameplayer*, you should update dependencies using the package manager yarn.

Next, push all your local changes to your own fork. You should push your code (making sure to replace *feature/my-validation-feature* with whatever your branch is actually called):

```
git push --set-upstream origin feature/my-validation-feature
```

Prior to finalizing your commit, make sure to clean up your code to comply with PEP8. Since both *black* and *isort* are included in our development dependencies, you should just be able to run `isort -rc . --skip venv` to fix your imports, and similarly `black . --exclude=venv` to “blacken” your changes. With both commands, replace *venv* with the actual name of your virtual env directory so that you don’t blacken/isort your dependencies.

When your branch is ready (you’ve tested your changes out, and your code has comments and tests), submit a Pull Request! To do this, go to GitHub, navigate to your fork (in this case the github extension should be */your-username/lookit-api*), then click *new pull request*. Change the base to *develop* and the compare to *feature/my-validation-feature*. Finally, click *Create pull request* and describe the changes you have made. Your pull request will be reviewed by Lookit staff; changes may be requested before changes are merged into the *develop* branch. To allow Lookit staff to add changes directly to your feature branch, follow the directions [here](#).

**IMPORTANT: WHEN YOUR PR IS ACCEPTED**, stop using your branch right away (or delete it altogether). New features (or enhanced versions of your existing feature) should be created on brand new branches (after pulling in all the fresh changes from *develop*).

## 6.5 Writing your tests

In *lookit-api*, you should generally add to or edit the *tests.py* file in the appropriate app (e.g., *exp/tests.py*). You can run tests like this:

```
python manage.py test
```

For more information see <https://docs.djangoproject.com/en/2.1/topics/testing/>.

In *ember-lookit-frameplayer* you should generally edit the tests under *tests/*, but as you will see there is currently very little coverage. Just try to leave it better than you found it.

In *ember-lookit-frameplayer*, you should generally add a test file under *tests/unit/components/* if you have created a new frame. As you can see, we do not have a strong convention for this yet except for randomizer frames.

To learn more about how testing is supposed to work for *ember-lookit-frameplayer*, see <https://guides.emberjs.com/v2.11.0/testing/>.

If you are editing the documentation, please don't write tests! Just actually try building it so you'll notice if something's not formatted the way you expected.

## 6.6 Editing the Lookit documentation

Documentation for use of the Lookit platform (what you're reading now!), including *both* the Django site `lookit-api` and the Ember application `ember-lookit-frameplayer` used for the actual studies, lives in the [lookit-docs repo](#) under `docs`. When you download this repo, there are two major folders to pay attention to: `build` and `source`, both of which are found in the `docs` folder. All editing should be done in the `source` folder, as the files in the `build` folder are Sphinx generated HTML files.

The file `index.rst` contains the table of contents (look for `toctree`). If you wish to add another page to the docs, you must add a new entry on the `toctree`. Documentation is written using [ReStructured Text \(RST\) markup](#). It is also possible to add Markdown (`.md`) files and have them included in the documentation, but for consistency we are trying to keep all documentation in `.rst` format. If you are more familiar with Markdown, you can convert between formats using [Pandoc](#), e.g.:

```
pandoc -o outputfile.rst inputfile.md
```

If you are making substantial changes, you will want to take a look at how those changes look locally by using Sphinx to build your own local copy of the documentation. To do this, first create another virtual environment and install the requirements for Sphinx there:

```
/lookit-docs $ virtualenv -p python3 env
/lookit-docs $ source env/bin/activate
(env) /lookit-docs $ pip install -r docs/requirements.txt
```

You can then build the docs from within the `docs` directory:

```
(env) /lookit-docs/docs $ make html
```

Open another terminal and activate `env` again, then start a python server:

```
(env) /lookit-docs $ python -m http.server
```

Navigate to your local host in ur browser to see the docs. Edit the docs in rST. In your editor, you will need to run `make html` every time you want to see the changes you made, as this command will update the Sphinx generated HTML files with the edits made in the rST.

To edit the documentation, please submit a PR to the `lookit-docs/develop` branch; when it's merged, the docs served by ReadTheDocs at <https://lookit.readthedocs.io> will be automatically updated! (Note that it is easy to have ReadTheDocs serve multiple versions of the documentation, from different branches; we just haven't reached the point of that being more useful than confusing yet.)

### 7.1 Children

Children belonging to Participants. Many Studies on Organization Sites involve testing the behavior of children. The Participant registering the child must be the child's parent or legal guardian. A Child can only belong to one Participant.

### 7.2 Demographic Data

When Participants create accounts on an Organization Site, they are asked to fill out a demographic survey. Demographic Data is versioned, so if Participants update their Demographic Data, a new version is saved. When Participants take part in a study, the latest version of Demographic Data is linked to their study Response.

### 7.3 Experimenter

Application where Researchers develop Studies, request to post Studies to an Organization Site, and access data collected.

### 7.4 Feedback

Researchers can leave Feedback to Participants via the API about a particular Response to a Study. The Participant can view this Feedback on the Past Studies page.

## 7.5 Groups

Each Organization and each Study will have one group for each set of permissions. Groups are an easy way to manage many permissions. These groups will be created automatically when an Organization or Study is created. For example, if we have an Organization called MIT, then there will be an MIT\_ORG\_ADMIN, MIT\_ORG\_READ, and MIT\_ORG\_RESEARCHER group. A Researcher belonging to any of these groups will inherit all of the permissions associated. Members of MIT\_ORG\_ADMIN can edit any Study in MIT, while MIT\_ORG\_READ members can only view Studies within MIT. Members of MIT\_ORG\_RESEARCHER can create Studies but can only view Studies they have specific permission to view.

When a Study is created, two permission groups will also be created for it. If you create a Study called “Apples”, and you belong to the MIT Organization, the groups created will be MIT\_APPLES\_<STUDY\_ID>\_STUDY\_ADMIN and MIT\_APPLES\_<STUDY\_ID>\_STUDY\_READ. The Study’s creator is automatically added to the Study’s admin group. Researchers belonging to a Study’s admin or Study’s read group will inherit the associated permissions to that Study.

## 7.6 Organization

An institution (e.g., Lookit) or lab that has been registered with Experimenter. Each Organization has its own interface (Organization Site) where Studies are posted. All Organizations’ data are separate. Each Organization has their own Researchers, admins, Studies, Participants, etc. You can only view data that you have permission to see (depending on your admin/read/researcher permissions), and only data within your Organization.

## 7.7 Organization Site

One instance of a front-end where studies are posted. (Example: lookit.mit.edu)

## 7.8 Participants

Account holders - registered Lookit users who can take part in studies. ‘Participant’ refers to the account (generally held by a parent) rather than the individual child.

## 7.9 Researchers

Individuals posting Studies, collecting data, or administrating Organization Sites.

## 7.10 Responses

When a Participant takes part in a study, the answers to their questions, as well as other metadata like the time taking the study, are saved in a Response. In addition, many Responses are associated with video attachments.

## 7.11 Study

An experiment posted to an Organization Site.



## 8.1 Internal Resources

### 8.1.1 Docker

Docker is used as an alternative to virtual machines. Docker Makefiles, or *Dockerfiles*, are recipes containing all dependencies and the library needed to build an *image*. An *image* is a template that is built and stored and acts as a model for *containers*, analogous to the class-object relationship in object-oriented programming. It contains the application plus the necessary libraries and binaries needed to build a *container*. Since they are templates, *images* are what can be shared when exchanging containerized applications. On the other hand, containers are an ephemeral running of a process, which makes it impossible to share them. Instances of the class, or objects, are to Class as a *container* is to an *image*. A *container* runs a program in the context of an *image*.<sup>\*0</sup>

Docker is software that allows you to create isolated, independent environments, much like a virtual machine. To understand how Docker accomplishes these feats, you must first understand both the PPID/PID hierarchy of the Linux kernel and union file systems.

When using the Linux OS, every program running on your machine as a process ID (PID) and a parent process ID (PPID). Some programs (parent programs) have the ability to launch other programs (child programs). The PID of the original program becomes the PPID of the child process. This system forms a tree of processes branching off of one another. These ID numbers correspond to which port the programs are running on.

The root node of this “process tree” is called `systemd`, a.k.a. system daemon. It has PID 1 and PPID 0. In older distributions of Linux, the `init` process was used.<sup>1</sup> The job of this process is to launch other processes and adopt any orphaned processes (child processes whose parent processes have been killed). All of these programs can interact with each other through shared memory and message passing method.

Shared memory is easily understood via the Producer, Consumer scenario. Imagine you have two people, a producer and a consumer. When the producer creates a good, it will put it in a store where the consumer can find and consume

---

<sup>0</sup> Docker has many other moving parts behind the scenes. An example of a part is volumes. Volumes serve as a storage space for your containers. For more in depth information about volumes, check out this [link](#) In addition, this [series of articles](#) covers a lot of Docker topics not mentioned in this documentation.

<sup>1</sup> If you’re interested in learning about the difference between `init` and `systemd` as well as the reasoning behind the switch, check out this [link](#).

it. This store is like shared memory.

The message passing method utilized communication links to connect two processes and allow them to send messages to each other. After a link is established, processes can send messages that contain a header and a body. The header contains the metadata, such as the type of message, recipient, length, etc.

These communication methods allow for processes to interact with each other, and, as you can imagine, this creates a problem when it comes to isolation.

that `systemd` is a daemon process which creates a global namespace, and child processes create their own nested namespaces within the context of their parent namespace.

Programs can interact with each other when they're in the same *namespace*, referred to as a `systemd` tree.<sup>2</sup> In other words, the `systemd` is a daemon process that creates a global namespace, and all programs that share the same `systemd` command can interact with each other. Child processes create their own `systemd`, or nested namespaces, withing the context of their parent namespace. Docker utilizes this to create a new namespace, a.k.a. a *userspace*. When the Docker process is run, it is launched from the real `systemd` process and given a normal PID. A new feature released by Linux in 2008 allows Docker to have more than one PID. With this new feature, the nested namespace comes with a table that can map a relative PID seen by your container in said nested namespace to the actual PID seen by your machine. This allows Docker to take on PID 1 in your container.

Now, all programs that stem from the Docker branch will see Docker in the same way they see the `systemd` process. It is the root node, and they will not leave the nested namespace, effectively cutting off interaction between those processes and the ones on the rest of the real `systemd` tree.

Though this is a step closer to isolation, it is not quite there yet. Even though processes can't interact with the main branch, they can still interact with the main filesystem. To combat this, Docker makes use of union filesystems.<sup>3</sup> A union filesystem uses union mounts to merge two directories into one. This means that if you have two directories that contain different files, a union mount will create a new directory that contains all of the files from both. If both directories contain a file of the same, the mount usually has a system in place for which one it will choose.

One big thing that makes this file system important for Docker is its deletion process. When you delete a file in a union filesystem, it does not actually delete it, rather it adds an extra layer and masks it. This masking process allows Docker to unionize your machine's filesystem and your Docker filesystem, masking all files that are specific to your machine. The necessary directories for Linux set up, such as the `/etc`, `/var`, `/`, `usr`, and `home` directories are still intact, however extra, added files from your machine will be masked from Docker. In addition, when you write files to this union filesystem, they will not be written to your machine's file system, effectively isolating your machine from your containers.

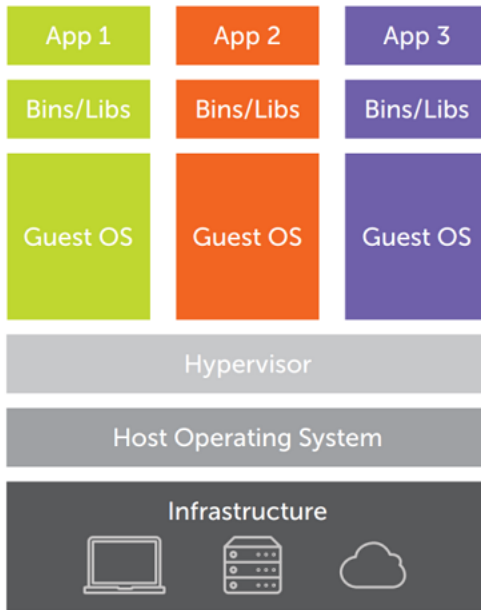
Another big difference between Docker and Virtual Machines is the Hypervisor.<sup>4</sup> When using a VM, a hypervisor is necessary to supervise, schedule, and control interactions between the host OS and the Guest OS. It acts as a layer of security between your machine and the virtual one so that yours is not damaged or messed with. Docker eliminates the need for the hypervisor because there is no longer a Guest OS. The Docker Engine is software downloaded directly onto your machine, and the containers run on the engine. Using Docker eliminates extra steps needed for the VM, as it doesn't have to virtualize an entire computer. This makes Docker faster, more efficient, and less redundant than VMs.

---

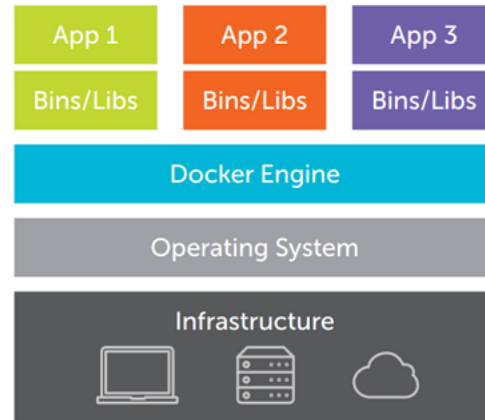
<sup>2</sup> The Linux kernel has many built in namespaces that are responsible for different things. If you are interested in learning more about this topic, check out this article on [namespaces](#)

<sup>3</sup> The union filesystem utilizes set theory. For a more in depth explanation of how they work and the math behind them, check out this article on [union filesystems](#)

<sup>4</sup> Hypervisors are essential to the functionality of VMs. If you want to know more about them, check out this link on [hypervisors](#)



Virtual Machines



Containers

image credits: [docker.com](https://www.docker.com/)

For a more in-depth explanation of Docker and how it works, consider looking at [this series of articles](#).

### 8.1.2 Postgres

PostgreSQL is a general purpose and object-relational database management system. It allows you to store and query large amounts of data without having to worry about all of the complicated processes going on under the hood.<sup>0</sup> PostgreSQL optimizes data querying for you, making your application faster. All information and metadata is stored in it.

### 8.1.3 RabbitMQ

RabbitMQ is a message broker. When messages are sent online, they go from the producer to a queue and then to the consumer. RabbitMQ is that queue. Instead of having to perform all tasks involving sending messages, including generating PDFs, locating the recipient, etc., the message producer only has to upload their message and instructions to the queue and RabbitMQ will take care of the rest. Using this service makes messaging through Lookit easier and more efficient, as it is able to re-queue messages, it is faster and more reliable, and it is scalable for when there are a lot of messages.

[source](#)

### 8.1.4 Ngrok

Ngrok is used in the development process to act as a tunnel into your PC. It is not secure to allow access into your PC or local address through a public channel, as this can open you up to malicious attacks. Ngrok allows you to securely provide access to your local address through something public, like the internet. When Ngrok is running on your

<sup>0</sup> add foot/endnote on what postgres is doing behind the scenes

machine, it gets the port of the network service, which is usually a web server, and then connects to the Ngrok cloud service. This cloud service takes in traffic and requests from a public address and then relays that traffic directly to the Ngrok process running on your machine, which then passes it along to the local address you specify. By doing this, you can minimize the interaction between outside traffic and your personal machine.

When trying to stream videos in the development stage, the ember frameplayer will stream the video to PIPE's web-servers, which will then write a request to AWS S3. A webhook will then send this video data back to the dummy url generated by ngrok, which then tunnels it to your local machine.

a dummy link for this purpose and then send the video from this dummy address to your PC.

data to AWS S3 instance. Webhook (allows us to say send a payload to a certain address when u finish doing this) sends request back to url generated by ngrok which serves as a local tunnel to your computer since u cant point a webhook to local host

then when the trigger finishes storing, it trips the "Finished uploading to S3" webhook that then sends a payload back to the lookit-API server, which has a handler that renames the file that was just stored in S3, among other things so that handler receives a payload from Pipe's "finished uploading to S3 webhook" i.e., it's Pipe telling lookit-API "hey, I finished putting some video in S3. Here's some identifying data about that video."

## 8.2 External Resources

### 8.2.1 Google Cloud

The Cloud service is where all the code for the studies and

### 8.2.2 Amazon Web Services

This is where al web-cam video recorded is stored

### 8.2.3 Celery

This is what runs the long term tasks

### 8.2.4 Authenticator

Allows you to log into your account securely

### 8.2.5 Lookit Ember Frameplayer

The frameplayer that provides the functionality for the experiments themselves. It parses the JSON documents specifying the study protocol into a sequence of "frames" and runs them. For more information, see *[Creating custom frames](#)*

### 8.2.6 PIPE

PIPE is a wrapper around the webRTC recorder that handles streaming. It is used to record the video and audio. WebRTC is what connects to the hardware of your computer and films for you. PIPE converts recorded files to ,mp4. <https://addpipe.com/about>

### 8.2.7 Footnotes

### 8.2.8 Endnotes