
Experimenter Documentation

Release 0.0.0

Center for Open Science

Oct 09, 2017

Contents

1	Using the staging server to try your experiments	3
1.1	A sandbox environment	3
1.2	Getting access	3
2	Building an Experiment	5
2.1	Prerequisites	5
2.2	Creating a new study and setting study details	5
2.3	Experiment structure	8
2.3.1	Nested randomizers	13
2.4	Testing your study	15
3	Experiment data	17
3.1	Accessing experiment data	17
3.2	Structure of session data	17
3.3	Sessions and <code>expData</code> in detail	19
4	Glossary of Experimental Components	21
4.1	general patterns	21
4.1.1	text-blocks	21
4.1.2	remote-resources	21
4.2	exp-audioplayer	22
4.2.1	example	22
4.2.2	parameters	23
4.2.3	data	23
4.3	exp-consent	23
4.3.1	example	24
4.3.2	parameters	24
4.3.3	data	24
4.4	exp-info	24
4.4.1	example	25
4.4.2	parameters	25
4.4.3	data	25
4.5	exp-survey	25
4.5.1	example	26
4.5.2	parameters	26
4.5.3	data	26
4.6	exp-video-config	27

4.6.1	example	27
4.6.2	parameters	28
4.6.3	data	28
4.7	exp-video-consent	28
4.7.1	example	28
4.7.2	parameters	30
4.7.3	data	30
4.8	exp-video-preview	30
4.8.1	parameters	30
4.8.2	data	31
5	Preparing your stimuli	33
5.1	Audio and video files	33
5.2	File formats	33
5.3	Directory structure	34
6	Development: Installation	37
6.1	Ember Dependencies	37
6.2	Other Dependencies	37
6.3	Installation	37
6.3.1	exp-addons Submodule	37
6.3.2	Javascript Dependencies	38
6.3.3	Add a .env file	38
6.3.4	Run the Ember server	39
6.3.5	Bootstrapping in Example Data	39
7	Development: Custom Frames	41
7.1	Overview	41
7.2	Getting Started	41
7.2.1	A Simple Example	42
7.2.2	Building out the Example	43
7.3	Tips and tricks	45
7.3.1	Tips for adding styles	45
7.3.2	When should I use actions vs functions?	45
8	Development: Mixins of premade functionality	47
8.1	FullScreen	47
8.2	MediaReload	47
8.3	VideoPause	47
8.4	VideoRecord	47
9	Development: Randomization	49
9.1	Overview of ‘choice’ structure	49
9.2	Making your own	49
10	How to capture video in an experiment	53
10.1	Troubleshooting	54
11	Architecture	55
11.1	JamDB as a Backend	55

NOTE: This documentation is a work in progress

Experimenter is a platform for designing and administering experiments. It is built on top of [JamDB](#) and [Ember.js](#), and is developed by the [Center for Open Science](#).

Contents:

Using the staging server to try your experiments

A sandbox environment

The staging versions of [Lookit](#) and [Experimenter](#) allow you to write and try out your studies, without posting them on the main site and collecting data. On Experimenter, you can create a new study, edit the details about it like the summary and age range, edit the JSON schema that defines what happens in the study, and start/stop data collection. Lookit accesses that data to show the studies that are currently collecting data and parses the study description. The staging-lookit application is separate from the production version of Lookit at lookit.mit.edu; account data isn't shared. Any data stored on staging-lookit is expected to be temporary test data.

Note: Technically, staging-lookit is public - anyone can create an account there, and see the studies being developed. It is, therefore, not a good place for your super-secret experimental design, working perpetual motion machine plans, etc. But in practice, no one's going to stumble on it.

It's also possible to run Lookit and/or Experimenter locally, so that you can edit the code that's used. In this case, they still talk to either the staging or the production server to fetch the definitions of available studies (Lookit) or save those definitions (Experimenter). For now, the plan is that you do NOT need to edit any of the code - if you want frames to work differently, in ways that aren't possible to achieve by adjusting the data you pass to them using the JSON schema, you'll contact MIT.

Getting access

To access staging-experimenter, you'll need an account on staging.osf.io. Once you've created an account, go to [your profile](#) and send MIT your 5-character profile ID from the end of your "public profile" link (e.g. staging.osf.io/72qxr) to get access to experimenter.

Once you can log in to experimenter, you may be prompted to select a namespace. Select 'lookit'.

Building an Experiment

Prerequisites

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

Additionally, we use JSON Schema heavily throughout this project. The examples [here](#) are a great place to learn more about this specification.

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

Creating a new study and setting study details

You can click ‘New Experiment’ to get started working on your own study, or clone an existing study to copy its experiment definition.

Here is the ‘experiment detail view’ for an example study. The primary purpose of the details you can edit in this view is to display the study to parents who might be interested in participating. You can select a thumbnail image, give a brief description of the study (like you would to a parent on the phone or at the museum if you were recruiting), and define an age range or other eligibility criteria. The “Participant Eligibility” string is just a description and can be in any format you want (e.g. “for girls ages 3 to 5” is fine) but parents will only see a warning about study eligibility based on the minimum/maximum ages you set. Those can be in months, days, or years. Parents who try to participate will see a warning if their child is younger (asking them to wait if they can) or older (letting them know we won’t be able to use their data) but are not actually prevented from participating.

You won’t see your study on Lookit until it’s started (made active). You can start/stop your study here on the detail page.



Experimenter

- [Experiments](#)
- [Project Settings](#)
- [Create users](#)
- [Logout](#)

Labels & concepts

[✎ Edit Details](#)



For testing labels & concepts frames (Bria & Mike). Here you would describe what happens during the study (a few sentences, very concrete - "In this study, your baby will see pictures of objects in a particular category (like staplers or hedgehogs), accompanied by either words or tones. Then ..."

Purpose:
Here you would describe what the point is - e.g., we're interested in how labels affect category formation, and here's why that's interesting!

Duration: 10 minutes (note: this is just an estimate, for the parent's information)

Exit URL: Not specified

Participant Eligibility: For babies from 6 to 12 months old (this is just a string, doesn't have to be in any particular format)

Minimum Age: 6 months

Maximum Age: 12 months

Last Edited: March 26, 2017

Status: Active
Stop Experiment Now



Build Experiment

Add/Modify experiment components



View 0 Responses

Inspect responses from studies



Clone Experiment

Copy experiment structure and details

Delete this Experiment
You can't delete active experiments

Here are the corresponding study views on Lookit:



Labels & concepts

For testing labels & concepts frames (Bria & Mike).
Here you would describe what happens during the
study (a few sentences, very concrete - "in this
study, your baby will see pictures of objects in a
particular category like shoes or hedgehogs").

[See details](#)



Thank you for your interest in this study! We'll help you learn more and get started.

[Log in to participate](#)

Eligibility criteria: For babies from 6 to 12 months old (this is just a string, doesn't have to be in any particular format)

Duration: 10 minutes (note: this is just an estimate, for the parent's information)

What happens: For testing labels & concepts frames (Bria & Mike). Here you would describe what happens during the study (a few sentences, very concrete - "in this study, your baby will see pictures of objects in a particular category (like staplers or hedgehogs), accompanied by either words or tones. Then ..."

What are we studying?: Here you would describe what the point is - e.g., we're interested in how labels affect category formation, and here's why that's interesting!

Try it yourself: Make your own study on staging-experimenter, choose a thumbnail, and enter a description. Look on Lookit: you don't see it, because you haven't started the study yet. Start the study from Experimenter and refresh Lookit: there it is!

Your study's unique ID can be seen in the URL as you view it from either Experimenter or Lookit.

Experiment structure

To define what actually happens in your study, go to “Build Experiment” at the bottom of the detail page. In this “experiment editor” view, Experimenter provides an interface to define the structure of an experiment using a JSON document. This is composed of two segments:

- **structure:** a definition of the **frames** you want to utilize in your experiment. This must take the form of a JSON object, i.e. a set of key/value pairs.
- **sequence:** a list of keys from the **structure** object. These need not be unique, and items from **structure** may be repeated. This determines the order that **frames** in your experiment will be shown.

Note: the term **frame** refers to a single segment of your experiment. Examples of this might be: a consent form, a survey, or some video stimulus. Hopefully this idea will become increasingly clear as you progress through this guide.

To explain these concepts, let's walk through an example:

```
{  
  "frames": {  
    "intro-video": {
```

```

    "kind": "exp-video",
    "sources": [
      {
        "type": "video/webm",
        "src": "https://s3.amazonaws.com/exampleexp/my_video.webm"
      },
      {
        "type": "video/ogg",
        "src": "https://s3.amazonaws.com/exampleexp/my_video.ogg"
      },
      {
        "type": "video/mp4",
        "src": "https://s3.amazonaws.com/exampleexp/my_video.m4v"
      }
    ]
  },
  "survey-1": {
    "formSchema": "URL:https://s3.amazonaws.com/exampleexp/survey-1.json",
    "kind": "exp-survey"
  },
  "survey-2": {
    "formSchema": "URL:https://s3.amazonaws.com/exampleexp/survey-2.json",
    "kind": "exp-survey"
  },
  "survey-3": {
    "formSchema": "URL:https://s3.amazonaws.com/exampleexp/survey-3.json",
    "kind": "exp-survey"
  },
  "survey-randomizer": {
    "options": [
      "survey-1",
      "survey-2",
      "survey-3"
    ],
    "sampler": "random",
    "kind": "choice"
  },
  "exit-survey": {
    "formSchema": "URL:https://s3.amazonaws.com/exampleexp/exit-survey.json",
    "kind": "exp-survey"
  }
],
"sequence": [
  "intro-video",
  "survey-randomizer",
  "exit-survey"
]
}

```

This JSON document describes a fairly simple experiment. It has three basic parts (see ‘sequence’):

1. intro-video: A short video clip that prepares participants for what is to come in the study. Multiple file formats are specified to support a range of web browsers.
2. survey-randomizer: A **frame** that randomly selects from one of the three ‘options’, in this case ‘survey-1’, ‘survey-2’, or ‘survey-3’. The `"sampler": "random"` setting tells Experimenter to simply pick of of the options at random. Other supported options are described [here](#).
3. exit-survey: A simple post-study survey. Notice for each of the **frames** with `"type": "exp-survey"`

there is a `formSchema` property that specifies the URL of another JSON schema to load. This corresponds with the input data expected by [Alpaca Forms](#). An example of one of these schemas is below:

```
{
  "schema": {
    "title": "Survey One",
    "type": "object",
    "properties": {
      "name": {
        "type": "string",
        "title": "What is your name?"
      },
      "favColor": {
        "type": "string",
        "title": "What is your favorite color?",
        "enum": ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]
      }
    }
  }
}
```

A Lookit study schema

A typical Lookit study might contain the following frame types:

1. exp-video-config
2. exp-video-consent
3. exp-lookit-text
4. exp-lookit-preview-explanation
5. exp-video-preview
6. exp-lookit-mood-questionnaire
7. exp-video-config-quality
8. exp-lookit-instructions
9. [Study-specific frames, e.g. exp-lookit-geometry-alternation, 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

For now, before any fullscreen frames, a frame that extends `exp-frame-base-unsafe` (like `exp-lookit-instructions`) needs to be used so that the transition to fullscreen works smoothly. A more flexible way to achieve this behavior is in the `works!`

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 can be automatically expanded to a list of frames, so that for instance you can specify your 12 looking-time trials all at once.

For complete documentation of available randomizers, see [<http://centerforopenscience.github.io/exp-addons/modules/randomizers.html>] (<http://centerforopenscience.github.io/exp-addons/modules/randomizers.html>).

To use a randomizer frame, you set `"kind"` to `"choice"` and `"sampler"` to the appropriate type of randomizer. The most common type of randomizer you will use is called `[random-parameter-set]` (<http://centerforopenscience.github.io/exp-addons/classes/randomParameterSet.html>).

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

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

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 study JSON schema:

```
{
 "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."
 }
],

```

```

 {
 "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
 }
],
 "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`.

If we wanted to have 75% of participants think about how tasty broccoli is, we could also weight the parameter sets by providing the optional parameter `"parameterSetWeights": [3, 1]` to the randomizer frame.

Note: One use of `parameterSetWeights` is to stop testing conditions that you already have enough children in as data collection proceeds.

## Nested randomizers

The frame list you provide to the `randomParameterSet` randomizer can even include other randomizer frames! This allows you to, for instance, define a **trial** that includes several distinct **blocks** (say, an intro, video, and then 4 test questions), then show 10 of those trials with different parameters - without having to write out all 60 blocks. There's nothing "special" about doing this, but it can be a little more confusing.

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.

```
"nested-trials": {
 "kind": "choice",
 "sampler": "random-parameter-set",
 "commonFrameProperties": {
 "kind": "choice",
 "sampler": "random-parameter-set",
 "frameList": [
 {
 "nPhase": 0,
 "doRecording": false,
 "autoProceed": false,
 "parentTextBlock": {
 "title": "Parents!",
 "text": "Phase 0: instructions",
 "emph": true
 },
 "images": [
 {
 "id": "protagonist",
 "src": "PROTAGONISTFACELEFT",
 "left": "40",
 "bottom": "2",
 "height": "60",
 "animate": "fadein"
 }
],
 "audioSources": [
 {
 "audioId": "firstAudio",
 "sources": [{"stub": "0INTRO"}]
 }
]
 },
 {
 "nPhase": 1,
 "doRecording": false,
 "autoProceed": false,
 "parentTextBlock": {
 "title": "Parents!",
 "text": "Phase 1: instructions",
 "emph": true
 },
 "images": [
 {
 "id": "protagonist",
 "src": "PROTAGONISTFACELEFT",
 "left": "40",
 "bottom": "2",
 "height": "60"
 }
],
 "audioSources": [
 {
 "audioId": "firstAudio",
 "sources": [{"stub": "1INTRO"}]
 }
]
 }
]
 }
}
```

```

]
 },
 "commonFrameProperties": {
 "kind": "exp-lookit-dialogue-page",
 "doRecording": true,
 "nTrial": "NTRIAL",
 "backgroundImage": "BACKGROUNDIMG",
 "baseDir": "https://s3.amazonaws.com/lookitcontents/politeness/",
 "audioTypes": ["mp3", "ogg"]
 },
},
"frameList": [
 {
 "parameterSets": [
 {
 "PROTAGONISTFACELEFT": "PROTAGONISTFACELEFT_1",
 "BACKGROUNDIMG": "BACKGROUNDIMG_1",
 "0INTRO": "0INTRO_1",
 "1INTRO": "1INTRO_1",
 "NTRIAL": 1
 }
]
 },
 {
 "parameterSets": [
 {
 "PROTAGONISTFACELEFT": "PROTAGONISTFACELEFT_2",
 "BACKGROUNDIMG": "BACKGROUNDIMG_2",
 "0INTRO": "0INTRO_2",
 "1INTRO": "1INTRO_2",
 "NTRIAL": 2
 }
]
 }
],
"parameterSets": [
 {
 "PROTAGONISTFACELEFT_1": "order1_test1_listener1.png",
 "PROTAGONISTFACELEFT_2": "order1_test1_listener1_second.png",
 "BACKGROUNDIMG_1": "order1_test1_background.png",
 "BACKGROUNDIMG_2": "order1_test1_background.png",
 "0INTRO_1": "polcon_example_1intro",
 "1INTRO_1": "polcon_example_1intro",
 "0INTRO_2": "polcon_example_1intro",
 "1INTRO_2": "polcon_example_1intro"
 }
]
}

```

## Testing your study

Experimenter has a built-in tool that allows you to try out your study. However, some functionality may not be exactly the same as on Lookit. We recommend testing your study from Lookit, which will be how participants experience it.



### Accessing experiment data

You can see and download collected data from sessions marked as ‘completed’ (user filled out the exit survey) directly from the Experimenter application.

You can also download JSON study or accounts data from the command line using the python script `experimenter.py` ([description](#)); you’ll need to set up a file `config.json` with the following content:

```
{
 "host": "https://staging-metadata.osf.io",
 "namespace": "lookit",
 "osf_token": "YOUR_OSF_TOKEN_HERE"
}
```

You can create an OSF token for the staging server [here](#).

The collection name to use to get study session records is `session[STUDYIDHERE]s`, e.g. `session58d015243de08a00400316e0s`.

### Structure of session data

The data saved when a subject participates in a study varies based on how that experiment is defined. The general structure for this **session** data is:

```
{
 "type": "object",
 "properties": {
 "profileId": {
 "type": "string",
 "pattern": "\\w+\\.\\w+"
 },
 "experimentId": {
```

```

 "type": "string",
 "pattern": "\\w+"
 },
 "experimentVersion": {
 "type": "string"
 },
 "completed": {
 "type": "boolean"
 },
 "sequence": {
 "type": "array",
 "items": {
 "type": "string"
 }
 },
 "conditions": {
 "type": "object"
 },
 "expData": {
 "type": "object"
 },
 "feedback": {
 "$oneOf": [{
 "type": "string"
 }, null]
 },
 "hasReadFeedback": {
 "$oneOf": [{
 "type": "boolean"
 }, null]
 },
 "globalEventTimings": {
 "type": "array",
 "items": {
 "type": "object"
 }
 }
},
"required": [
 "profileId",
 "experimentId",
 "experimentVersion",
 "completed",
 "sequence",
 "expData"
]
}

```

And descriptions of these properties are enumerated below:

- *profileId*: This unique identifier of the participant. This field follows the form: <account.id>.<profile.id>, where <account.id> is the unique identifier of the associated account, and <profile.id> is the unique identifier of the profile active during this particular session (e.g. the participating child). Account data is stored in a separate database, and includes demographic survey data and the list of profiles associated with the account.
- *experimentId*: The unique identifier of the study the subject participated in.
- *experimentVersion*: The unique identifier of the version of the study the subject participated in. TODO: more

on JamDB, versioning

- *completed*: A true/false flag indicating whether or not the subject completed the study.
- *sequence*: The sequence of **frames** the subject actually saw (after running randomization, etc.)
- *conditions*: For randomizers, this records what condition the subject was assigned
- *expData*: A JSON object containing the data collected by each **frame** in the study. More on this to follow.
- *feedback*: Some researchers may have a need to leave some session-specific feedback for a subject; this is shown to the participant in their ‘completed studies’ view.
- *hasReadFeedback*: A true/false flag to indicate whether or not the given feedback has been read.
- *globalEventTimings*: 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

```
"globalEventTimings": [
 {
 "exitType": "browserNavigationAttempt",
 "eventType": "exitEarly",
 "lastPageSeen": 0,
 "timestamp": "2016-11-28T20:00:13.677Z"
 }
]
```

## Sessions and expData in detail

Lets walk through an example of data collected during a session (note: some fields are hidden):

```
{
 "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?"
 },
 "conditionNum": 0
 }
 },
 "expData": {
 "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"
 }]
 }
 }
 }
}

```

Things to note:

- ‘sequence’ has resolved to three items following 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. Notice in particular that since ‘survey-2’ was randomly selected, it appears here.
- ‘conditions’ has the key/value pair `"1-survey": 1`, where 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). The data stored by a particular randomizer can be found under `methods: conditions` in the [randomizer documentation](#)
- ‘expData’ is an object with three properties (corresponding with the values from ‘sequence’). Each of these objects 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](#). Other properties besides ‘eventTimings’ are dependent on the **frame** type. You can see which properties a particular frame type records by looking at the parameters of the `serializeContent` method under the ‘Methods’ tab in its [frame documentation](#). Notice that ‘exp-video’ captures no data, and that both ‘exp-survey’ **frames** capture a ‘formData’ object.



---

## Glossary of Experimental Components

---

For the most current documentation of individual frames available to use, please see <http://centerforopenscience.github.io/exp-addons/modules/frames.html> and <http://centerforopenscience.github.io/exp-addons/modules/randomizers.html>.

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.

The below documentation may be out-of-date.

### general patterns

#### text-blocks

Many of these components expect one or more parameters with the structure:

```
{
 "title": "My title",
 "text": "Some text here", // optional
 "markdown": "Some markdown text here" // optional
}
```

This pattern will be referred to in the rest of this document as a ‘text-block’. Note: a text-block may specify *either* a ‘text’ property or a ‘markdown’ property. The ‘text’ property supports lightweight formatting using the ‘\n’ character to create a newline, and ‘\t’ to create indentation. The ‘markdown’ property will be formatted as [markdown](#).

#### remote-resources

Some items support loading additional content from a remote resource. This uses the syntax:

(JSON|URL) :<url>

where the `JSON:` prefix means the fetched content should be parsed as JSON, and the `URL:` prefix should be interpreted as plain text. Examples are:

```
"formSchema": "JSON:https://s3.amazonaws.com/exampleexp/my_survey.json"
```

and

```
"text": "URL:https://s3.amazonaws.com/exampleexp/consent_text.txt"
```

## exp-audioplayer

Play some audio for the participant. Optionally some some images while the audio is playing.

[view source code](#)

### example

#### Example audioplayer

##### A subheading

Some subheading text



##### Another prompt

Some text for another prompt

Please play the sample audio before continuing.

Continue

```
{
 "kind": "exp-audioplayer",
 "autoplay": false,
 "fullControls": true,
 "mustPlay": true,
 "images": [],
 "prompts": [{
 "title": "Instead of a consent form...",
 "text": "Here's a helpful tip."
 }, {
 "title": "A horse is a horse",
 "text": "But please don't say that backwards."
 }],
 "sources": [{
 "type": "audio/ogg",
 "src": "horse.ogg"
 }]
}
```

## parameters

- **autoplay**: whether to autoplay the audio on load
  - type: true/false
  - default: true
- **fullControls**: whether to use the full player controls. If false, display a single button to play audio from the start.
  - type: true/false
  - default: true
- **mustPlay**: should the participant be forced to play the clip before leaving the page?
  - type: true/false
  - default: true
- **sources**: list of objects specifying audio src and type
  - type: list
  - default: empty
- **title**: a title to show at the top of the frame
  - type: text
  - default: empty
- **titlePrompt**: a title and description to show at the top of the frame
  - type: text-block
  - default empty
- **images**: a list of objects specifying image src, alt, and title
  - type: list
  - default: empty
- **prompts**: text of any header/prompt paragraphs to show the participant
  - type: list of text-blocks
  - default: empty

## data

- **didFinishSound**: did the use play through the sound all of the way?
    - type: true/false
- 

## exp-consent

A simple consent form. Forces the participant to accept before continuing.

[view source code](#)

## example

### Just checking

Are you sure you know what you're getting into?

I'm sure. ☐

Continue

```
{
 "kind": "exp-consent",
 "title": "Just checking",
 "body": "Are you sure you know what you're getting into?",
 "consentLabel": "I'm sure."
}
```

## parameters

- **title**: a title for the consent form
  - type: text
  - default: ‘Notice of Consent’
- **body**: body text for the consent form
  - type: text
  - default: ‘Do you consent to take this experiment?’
- **consentLabel**: a label next to the consent form checkbox
  - type: text
  - default: ‘I agree’

## data

- **consentGranted**: did the participant grant consent?
  - type: true/false

## exp-info

Show some text instructions to the participant.

[view source code](#)

## example

### For yor information

#### Example 1.

This is just an example.

#### Example 2.

And this is another example

Next

```
{
 "kind": "exp-info",
 "title": "For yor information",
 "blocks": [{
 "title": "Example 1.",
 "text": "This is just an example."
 }, {
 "title": "Example 2.",
 "text": "And this is another example"
 }]
}
```

## parameters

- **title**: a title to go at the top of this block
  - type: text
  - default: empty
- **blocks**: a list of text-blocks to show
  - type: list of text-blocks
  - default: empty

## data

None

## exp-survey

Presents the participant with a survey.

[view source code](#)

## example

Survey One

What is your name?

What is your favorite color?

Continue

```
{
 "kind": "exp-survey",
 "formSchema": {
 "schema": {
 "type": "object",
 "properties": {
 "name": {
 "type": "string",
 "title": "What is your name?"
 },
 "favColor": {
 "type": "string",
 "enum": [
 "red",
 "orange",
 "yellow",
 "green",
 "blue",
 "indigo",
 "violet"
],
 "title": "What is your favorite color?"
 }
 },
 "title": "Survey One"
 }
 }
}
```

## parameters

- **formSchema**: a JSON Schema defining a form; uses [Alpaca Forms](#)
  - type: object (JSON Schema) or remote-resource
  - default: empty

## data

- **formData**: an object mapping to the properties defined in **formSchema**
  - type: object

## exp-video-config

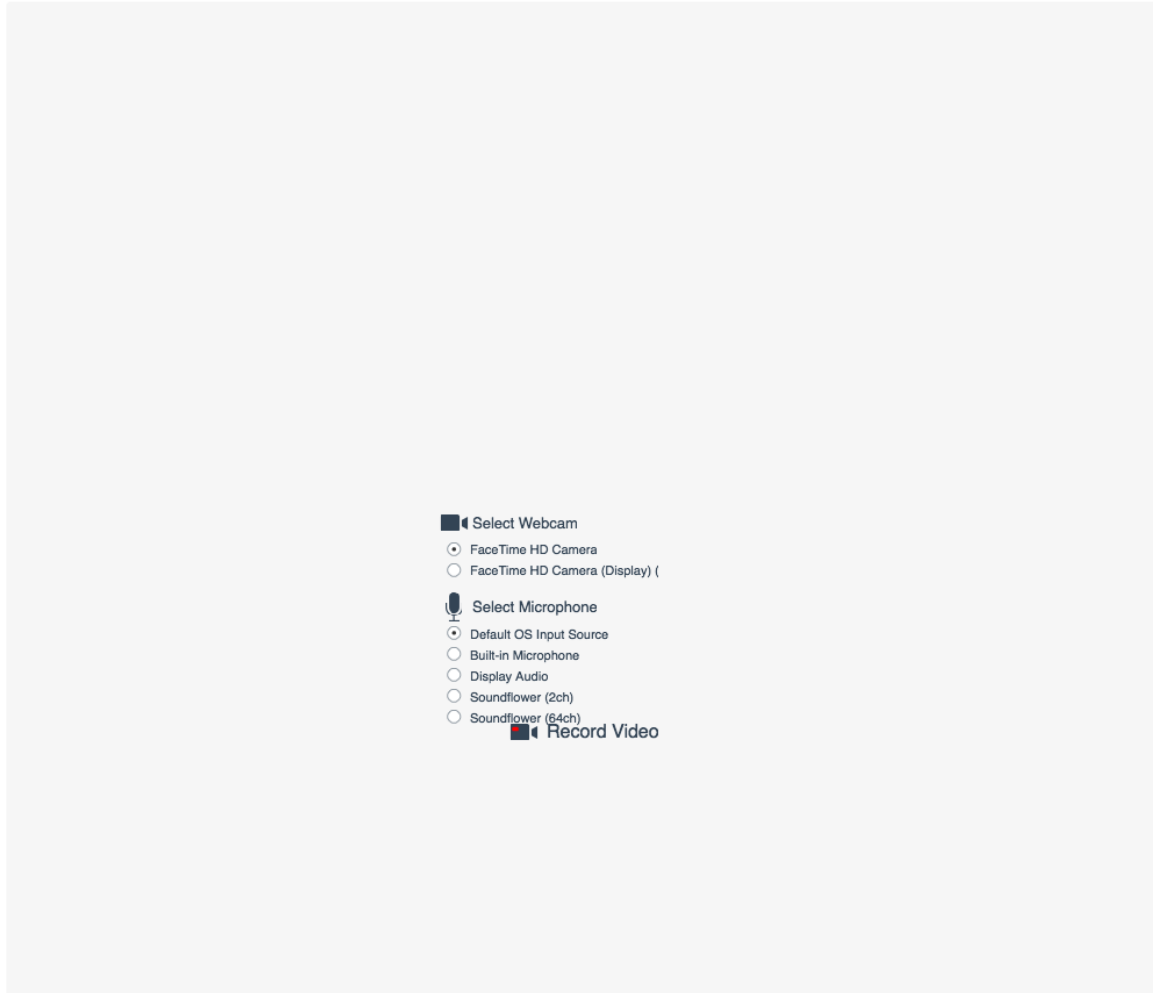
Help guide the participant through setting up her webcam.

[view source code](#)

### example

Please make sure your webcam and microphone are functioning correctly.

\* Please note: we are **not** recording any video during this setup.



The screenshot shows a configuration window with a light gray background. In the center, there are two sections: 'Select Webcam' and 'Select Microphone'. The 'Select Webcam' section has two radio button options: 'FaceTime HD Camera' (selected) and 'FaceTime HD Camera (Display)'. The 'Select Microphone' section has five radio button options: 'Default OS Input Source' (selected), 'Built-in Microphone', 'Display Audio', 'Soundflower (2ch)', and 'Soundflower (64ch)'. Below these sections is a 'Record Video' button with a red icon. At the bottom of the window is a 'Next' button.

```
{
 "kind": "exp-video-config",
 "instructions": "Please make sure your webcam and microphone are functioning_
↪correctly."
}
```

## parameters

- **instructions:** some instructions to show the participant
  - type: text
  - default: ‘Configure your video camera for the upcoming sections. Press next when you are finished.’

## data

None

---

## exp-video-consent

present the participant with a written consent document then capture her spoken consent

[view source code](#)

## example

If you'd like to participate, please read through the following information and then record your consent statement.

### Notice of consent

**Introduction**  
The purpose of this study is to learn about ...

**Privacy**  
We will not share your personal information with anyone.

Continue



I give my consent to participate in this study



Record

```
{
 "kind": "exp-video-consent",
 "prompt": "I give my consent to participate in this study",
 "blocks": [{
 "text": "The purpose of this study is to learn about ...",
 "title": "Introduction"
 }, {
 "text": "We will not share your personal information with anyone.",
 "title": "Privacy"
 }],
 "title": "Notice of consent"
}
```

## parameters

- **title**: title of written consent
  - type: text
  - default: ‘Notice of Consent’
- **blocks**: text-blocks of written consent
  - type: list of text-blocks
  - default: []
- **prompt**: a prompt to show for spoken consent
  - type: text
  - default: ‘I consent to participate in this study’

## data

- **videoId**: this unique id of the captured video
    - type: text
- 

## exp-video-preview

### parameters

- **index**: the zero-based index of the first video to show
  - type: number
  - default: 0
- **videos**: a list of videos to preview
  - type: list of objects with a src and type property
  - default: []
- **prompt**: Require a button press before showing the videos
  - type: text
  - default: empty
- **text**: Text to display to the user
- **type**: text-block
- **default**: Empty

**data**

None



---

## Preparing your stimuli

---

### Audio and video files

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

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.

### 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 the command-line tool [ffmpeg](#). It's a bit of a pain to get used to, but then you can do almost anything you might want to with audio and video files.

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')])
```

## Directory structure

For convenience, several of the newer frames allow 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`).

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

**Audio/video sources:** you will be providing a list of objects describing the source, 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 source like this:

```
[
 {
 "stub": "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 [{"stub": "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>.





### Ember Dependencies

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

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

### Other Dependencies

- [JamDB](#): note this requires Python 3.5.X

### Installation

First:

```
git clone https://github.com/CenterForOpenScience/experimenter.git
cd experimenter
```

### exp-addons Submodule

The exp-addons submodule allows for sharing some of the core Ember code for Experimenter's frontend between different apps. In particular it contains:

- **exp-player**: The build-in rendering engine for Experimenter
- **exp-models**: The ember-data models, adapters, serializers, authorizers, and authenticators used by Experimenter

To pull in the submodule, run:

```
git submodule init
git submodule update
```

## Javascript Dependencies

To install, run:

```
npm install
bower install

cd lib/exp-player
npm install
bower install

cd ../exp-models
npm install
bower install
```

## Add a .env file

This project needs a file named ‘.env’ in its root directory. This contains settings that are not suitable for publishing to GitHub. Your .env should look like:

```
OSF_CLIENT_ID=<client_id>
OSF_SCOPE=osf.users.all_read
OSF_URL=https://staging.osf.io
OSF_AUTH_URL=https://staging-accounts.osf.io

JAMDB_URL=https://staging-metadata.osf.io

WOWZA_PHP='{ } '
WOWZA_ASP='{ } '
```

These variables correspond with:

- **OSF\_CLIENT\_ID**: The client ID of a developer app created on the OSF. For development purposes please use: <https://staging.osf.io/settings/applications/>. Configure your app like:

**Application name**

**Project homepage URL**

**Application description**

**Authorization callback URL**

- **OSF\_SCOPE:** The “scope” determines what privileges will be required on behalf of that OSF user in order for experimenter to function. We do not recommend changing the default value.
- **OSF\_URL:** The URL of the OSF server you want to refer to. For develop please use our staging server.
- **OSF\_AUTH\_URL:** The URL of the OSF authentication server you wish to use. For development purposes please leave this pointed at the staging-accounts server.
- **JAMDB\_URL:** JamDB is the backend for Experimenter, and this URL determines which instance of the app your copy of Experimenter will use. For development purposes, please use the staging-metadata server.
- **WOWZA\_PHP/ASP:** These settings configure how the app will connect to a Wowza server (for streaming video uploads). Most developers will not need this feature, and if you believe you do please open an issue on our GitHub page: <https://github.com/CenterForOpenScience/experimenter/issues>.

## Run the Ember server

Run:

```
ember server
```

to fire up Ember’s built in server to test the app locally.

## Bootstrapping in Example Data

First, create a file `admins.json` in `experimenter/scripts/`:

```
[
 "<your_osf_id>"
]
```

Where `<your_osf_id>` is the user id of your [staging OSF Account](#).

Next, run:

```
npm run bootstrap
```

**Note:** this command will only work if you have successfully installed JamDB in a virtualenv named ‘jam’

Which will create:

- an ‘experimenter’ namespace
- all of the collections needed for Experimenter to work

---

## Development: Custom Frames

---

### Overview

You may find you have a need for some experimental component is not included in Experimenter already. 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.

Experimenter is composed of three main modules:

- **experimenter**: the main Experimenter GUI
- **lib/exp-models**: the data models used by **experimenter** and other applications
- **exp-player**: the built-in rendering engine for experiments built in Experimenter

Generally, all ‘frame’ development will happen in the exp-player module. By nature of the way the Experimenter repository is structured, this will mean making changes in the `experimenter/lib/exp-player` directory. These changes can be committed as part of the `exp-addons` git submodule (installed under `experimenter/lib`)

### 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 experimenter/lib/exp-player
ember generate exp-frame exp-<your_name>
```

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

## 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({
 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 comprised 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.

## 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">

 {{ consentLabel }}

 {{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')
});
```

## Tips and tricks

### Tips for adding styles

You will probably want to add custom styles to your frame, in order to control the size, placement, and color of elements. Experimenter uses 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 `addon/styles/components` directory of `exp-addons`. Then add a line to the top of `addon/styles/addon.scss`, telling it to use that style. For example,

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

Remember that anything in `exp-addons` is shared code. Below are a few good tips to help your addon stay isolated and distinct, so that it does not affect other projects.

Here are a few tips for writing good styles:

- Do not override global styles, or things that are part of another component. For example, `exp-video-physics` should not contain styles for `exp-player`, nor should it
  - If you need to style a button specifically inside that component, either add a second style to the element, or consider using nested [CSS selectors](#).
- Give all of the styles in your component a unique common name prefix, so that they don't inadvertently overlap with styles for other things. For example, instead of `some-video-widget`, consider a style name like `exp-myframe-video-widget`.

\* 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).

### 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)

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](#)

---

### Development: Mixins of premade functionality

---

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*. Below is a brief introduction to each of the common mixins; see sample usages throughout the exp-addons codebase. More documentation may be added in the future.

#### 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 at least once before full screen mode can become active.

#### 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.

#### VideoPause

Functionality related to pausing a video when the user presses the spacebar.

#### VideoRecord

Functionality related to video capture, in conjunction with the HDFVR/ Wowza system (for which MIT has a license).



---

## Development: Randomization

---

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. The goal of this page is to walk through an example of implementing a custom ‘randomizer’.

### 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)

### Making your own

There is some template code included to help you get started. From within the `experimenter/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];
 }
 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)
);

```





---

## How to capture video in an experiment

---

The Experimenter platform provides a means to capture video of a participant during an experiment, using the computer's webcam.

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
});
```

Within that frame, you will need some boilerplate that decides how to activate the recorder, and when to start recording. Below is an example from `exp-video-physics`, which starts recording immediately, and makes a copy of the recorder available on the component so that you can use additional helper methods (like `show`, `hide`, and `getTime`) as appropriate to deal with recording problems.

```
didInsertElement() {
 this._super(...arguments);

 if (this.get('experiment') && this.get('id') && this.get('session') && !this.
↪get('isLast')) {
 let recorder = this.get('videoRecorder').start(this.get('videoId'), this.
↪$('#videoRecorder'), {
 hidden: true
 });
 recorder.install({
 record: true
 }).then(() => {
 this.sendTimeEvent('recorderReady');
 this.set('recordingIsReady', true);
 });
 recorder.on('onCamAccess', (hasAccess) => {
 this.sendTimeEvent('hasCamAccess', {
```

```

 hasCamAccess: hasAccess
 });
 });
 recorder.on('onConnectionStatus', (status) => {
 this.sendTimeEvent('videoStreamConnection', {
 status: status
 });
 });
 this.set('recorder', recorder);
 },
},

```

Make sure to stop the recording when the user leaves the frame! You can ask the user to do this manually, or it can be done automatically via the following:

```

willDestroyElement() { // remove event handler
 // Whenever the component is destroyed, make sure that event handlers are
 ↪removed and video recorder is stopped
 if (this.get('recorder')) {
 this.get('recorder').hide(); // Hide the webcam config screen
 this.get('recorder').stop();
 }

 this.sendTimeEvent('destroyingElement');
 this._super(...arguments);
 // Todo: make removal of event listener more specific (in case a frame comes
 ↪between the video and the exit survey)
 $(document).off('keyup');
}

```

## Troubleshooting

If you are building a new ember app on this platform from scratch, you will need to do some setup to make video recording work. Because this relies on licensed software, it is not part of the default repository to be checked out.

1. Video recorder flash plugin (HDFVR): This is a Flash plugin that handles video recording. It requires a license; MIT can provide both the license and the required file. See README.md install instructions for details.
2. Config string: MIT must provide you with the values of `WOWZA_PHP=' {} '` and `WOWZA_ASP=' {} '` that you will need to place inside your `.env` file (as described in the project **README** file). This describes a WOWZA server backend that has been configured to receive and process video clips.
3. If you encounter problems finding the HDFVR video plugin, you may need to add the following markup to your `index.html` file: `<base href="/" />`

### JamDB as a Backend

Experimenter's backend is driven by [JamDB](#), which provides a performant API over a document store. It offers [JSON Schema](#) validation of submitted documents and a versioned history of all documents. This goal of this section of the documentation is to give a brief overview of the JamDB API in its relation to Experimenter.

For now, the best resource on learning more about this is reading the code [here](#) as well as the [JamDB documentation](#).