rovercode-web Documentation

Release 0.4.0

rovercode.com

Sep 27, 2017

Contents

License GPLv3

Source https://github.com/aninternetof/rovercode-web

Hosted at https://rovercode.com (master) and https://beta.rovercode.com (development)

CHAPTER 1

Welcome!

rovercode is an easy-to-use system for controlling robots (rovers) that can sense and react to their environment. The Blockly editor makes it easy to program and run your bot straight from your browser. Just drag and drop your commands to drive motors, read values from a variety of supported sensors, and see what your rover sees with the built in webcam viewer.

CHAPTER 2

Architecture

rovercode is made up of two parts:

- rovercode-web (the docs you're reading right now) is the web app running at rovercode.com.
- rovercode (a separate repo documented here) is the service that runs on the rover.

rovercode runs on the rover. The rover can be any single-board-computer supported by the Adafruit Python GPIO wrapper library, including the NextThingCo CHIP, Raspberry Pi, and BeagleBone Black.

rovercode-web is hosted on the Internet at rovercode.com. It has a Blockly-based editor (which we call Mission Control) for creating a routine. The routine executes in the browser (sandboxed, of course), and commands are sent to the rover for rovercode to execute (e.g. "stop motor, turn on light"). Events on the rover ("right eye detects something") are sent to the browser via a WebSocket connection.

The rover and the device running the browser must be on the same local network.

CHAPTER 3

Get Started

Check out the quickstart guide. Then see how to contribute.

CHAPTER 4

Contact

Please join the rovercode developer mailing list! Go here, then click “register”.

Also, we’d love to chat with you! Join the the rovercode Slack channel.

You can also email brady@rovercode.com.

CHAPTER 5

Contents

quickstart

Install docker and docker-compose, then

```
$ git clone --recursive https://github.com/aninternetof/rovercode-web.git && cd␣
↪rovercode-web
$ sudo docker-compose -f dev.yml build
$ sudo docker-compose -f dev.yml up
$ google-chrome localhost:8000
```

Basic Commands

rovercode-web runs is built with Django. During development, you can do regular Django things like this:

```
$ docker-compose -f dev.yml run django python manage.py migrate
$ docker-compose -f dev.yml run django python manage.py createsuperuser
```

If anything gives you trouble, see the detailed cookiecutter-django Docker documentation.

detailed usage

using rovercode with a rovercode-web hosted somewhere other than rovercode.com

By default, when rovercode runs, it registers itself with *https://rovercode.com*. But what if you want to try your changes to rovercode with *https://beta.rovercode.com*? Or with your local instance of rovercode-web (as described in the next section)? You can specify the target rovercode-web url by creating a .env file in your rovercode directory.

```
# first, navigate to the rovercode root diretory (same level as the Dockerfile), then
$ echo ROVERCODE_WEB_URL=https://beta.rovercode.com/ > .env
```

When you start rovercode, it will register itself with *beta.rovercode.com*.

develop rovercode and rovercode-web on the same machine at the same time

Get, build, and bring up rovercode-web as usual:

```
$ git clone --recursive https://github.com/aninternetof/rovercode-web.git && cd␣
↪rovercode-web
$ sudo docker-compose -f dev.yml build
$ sudo docker-compose -f dev.yml up
$ google-chrome localhost:8000
```

Get and build rovercode as usual:

```
$ git clone --recursive https://github.com/aninternetof/rovercode.git && cd rovercode
$ sudo docker build -t rovercode .
```

Set the url of the rovercode-web target to *http://rovercodeweb:8000*. You will see in the next step that this is the hostname that we assign to our local rovercode-web container.

```
# first, navigate to the rovercode root diretory (same level as the Dockerfile), then
$ echo ROVERCODE_WEB_URL=http://rovercodeweb:8000/ > .env
```

Finally, when you bring up the rovercode container, add a *link* flag to allow access between this container and your rovercode-web container.

```
$ sudo docker run -t --link rovercodeweb_django_1:rovercodeweb --net rovercodeweb_
↪default --name rovercode -v $PWD:/var/www/rovercode -p 80:80 -d rovercode
```

docker-compose named it *rovercodeweb_django_1*, but notice that we used a colon to rename it simply *rovercodeweb*. This is necessary, because this becomes the hostname, and Django does not like underscores in hostname headers.

We also had to add a *net rovercodeweb_default* flag, because docker-compose put rovercode-web on its own network instead of on the default one. (If you're curious, you can find its name using the command *sudo docker network ls*.)

rovercode is now running, and you can see that it has registered itself with your local rovercodeweb container by going to http://localhost:8000/mission-control/rovers. You can now select this rover in the mission-control interface, and rover commands will be sent to your rovercode container.

Attribution DEIS blog post

detailed usage

using rovercode with a rovercode-web hosted somewhere other than rovercode.com

By default, when rovercode runs, it registers itself with *https://rovercode.com*. But what if you want to try your changes to rovercode with *https://beta.rovercode.com*? Or with your local instance of rovercode-web (as described in the next section)? You can specify the target rovercode-web url by creating a .env file in your rovercode directory.

```
# first, navigate to the rovercode root diretory (same level as the Dockerfile), then
$ echo ROVERCODE_WEB_URL=https://beta.rovercode.com/ > .env
```

When you start rovercode, it will register itself with *beta.rovercode.com*.

develop rovercode and rovercode-web on the same machine at the same time

Get, build, and bring up rovercode-web as usual:

```
$ git clone --recursive https://github.com/aninternetof/rovercode-web.git && cd␣
↪rovercode-web
$ sudo docker-compose -f dev.yml build
$ sudo docker-compose -f dev.yml up
$ google-chrome localhost:8000
```

Get and build rovercode as usual:

```
$ git clone --recursive https://github.com/aninternetof/rovercode.git && cd rovercode
$ sudo docker build -t rovercode .
```

Set the url of the rovercode-web target to *http://rovercodeweb:8000*. You will see in the next step that this is the hostname that we assign to our local rovercode-web container.

```
# first, navigate to the rovercode root diretory (same level as the Dockerfile), then
$ echo ROVERCODE_WEB_URL=http://rovercodeweb:8000/ > .env
```

Finally, when you bring up the rovercode container, add a *link* flag to allow access between this container and your rovercode-web container.

```
$ sudo docker run -t --link rovercodeweb_django_1:rovercodeweb --net rovercodeweb_
↪default --name rovercode -v $PWD:/var/www/rovercode -p 80:80 -d rovercode
```

docker-compose named it *rovercodeweb_django_1*, but notice that we used a colon to rename it simply *rovercodeweb*. This is necessary, because this becomes the hostname, and Django does not like underscores in hostname headers.

We also had to add a *net rovercodeweb_default* flag, because docker-compose put rovercode-web on its own network instead of on the default one. (If you're curious, you can find its name using the command *sudo docker network ls*.)

rovercode is now running, and you can see that it has registered itself with your local rovercodeweb container by going to http://localhost:8000/mission-control/rovers. You can now select this rover in the mission-control interface, and rover commands will be sent to your rovercode container.

Attribution DEIS blog post

contribute

There is lots of fun work to be done!

Head on over to the rovercode-web github. We use ZenHub to improve GitHub's agile management, so install it, then visit the *boards* tab to find a fun card in the backlog. Or submit a new card for a bug or cool new feature idea.

And remember, you can do all these same things for rovercode.

Chat with us on the rovercode Slack channel.

Follow the the code of conduct.

modules

rovercode-web has two modules. rovercode-web serves the homepage and manages users. mission-control is the Blockly app.

mission-control

This is the web application where students create and run Blockly designs.

mission_control.apps	Mission Control apps.
mission_control.models	Mission Control models.
mission_control.serializers	Mission Control serializers.
mission_control.urls	
mission_control.utils	
mission_control.views	

mission_control.apps

Mission Control apps.

Classes

AppConfig(app_name, app_module)	Class representing a Django application and its configuration.
MissionControlConfig(app_name, app_module)	Configuration for the Mission Control app.

class mission_control.apps.**MissionControlConfig**(*app_name*, *app_module*)

Bases: django.apps.config.AppConfig

Configuration for the Mission Control app.

_path_from_module(*module*)

Attempt to determine app's filesystem path from its module.

check_models_ready()

Raises an exception if models haven't been imported yet.

create(*entry*)

Factory that creates an app config from an entry in INSTALLED_APPS.

get_model(*model_name*)

Returns the model with the given case-insensitive model_name.

Raises LookupError if no model exists with this name.

get_models(*include_auto_created=False*, *include_swapped=False*)

Returns an iterable of models.

By default, the following models aren't included:

- auto-created models for many-to-many relations without an explicit intermediate table,
- models created to satisfy deferred attribute queries,

•models that have been swapped out.

Set the corresponding keyword argument to True to include such models. Keyword arguments aren't documented; they're a private API.

ready()
Override this method in subclasses to run code when Django starts.

mission_control.models

Mission Control models.

Classes

BlockDiagram(*args, **kwargs)	Attributes to describe a single block diagram.
Rover(*args, **kwargs)	Attributes to describe a single rover.
User(id, password, last_login, is_superuser, ...)	

class mission_control.models.**BlockDiagram**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Attributes to describe a single block diagram.

_check_field_name_clashes()
Ref #17673.

_check_fields(***kwargs*)
Perform all field checks.

_check_id_field()
Check if *id* field is a primary key.

_check_index_together()
Check the value of "index_together" option.

_check_long_column_names()
Check that any auto-generated column names are shorter than the limits for each database in which the model will be created.

_check_m2m_through_same_relationship()
Check if no relationship model is used by more than one m2m field.

_check_managers(***kwargs*)
Perform all manager checks.

_check_ordering()
Check "ordering" option – is it a list of strings and do all fields exist?

_check_swappable()
Check if the swapped model exists.

_check_unique_together()
Check the value of "unique_together" option.

_do_insert(*manager*, *using*, *fields*, *update_pk*, *raw*)
Do an INSERT. If update_pk is defined then this method should return the new pk for the model.

_do_update(*base_qs*, *using*, *pk_val*, *values*, *update_fields*, *forced_update*)
This method will try to update the model. If the model was updated (in the sense that an update query was done and a matching row was found from the DB) the method will return True.

_get_unique_checks(*exclude=None*)
Gather a list of checks to perform. Since validate_unique could be called from a ModelForm, some fields may have been excluded; we can't perform a unique check on a model that is missing fields involved in that check. Fields that did not validate should also be excluded, but they need to be passed in via the exclude argument.

_save_parents(*cls*, *using*, *update_fields*)
Saves all the parents of cls using values from self.

_save_table(*raw=False*, *cls=None*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Does the heavy-lifting involved in saving. Updates or inserts the data for a single table.

clean()
Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

clean_fields(*exclude=None*)
Cleans all fields and raises a ValidationError containing a dict of all validation errors if any occur.

full_clean(*exclude=None*, *validate_unique=True*)
Calls clean_fields, clean, and validate_unique, on the model, and raises a `ValidationError` for any errors that occurred.

get_deferred_fields()
Returns a set containing names of deferred fields for this instance.

refresh_from_db(*using=None*, *fields=None*)
Reloads field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Handles the parts of saving which should be done only once per save, yet need to be done in raw saves, too. This includes some sanity checks and signal sending.

The 'raw' argument is telling save_base not to save any parent models and not to do any changes to the values before save. This is used by fixture loading.

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

validate_unique(*exclude=None*)
Checks unique constraints on the model and raises `ValidationError` if any failed.

class `mission_control.models.`**Rover**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Attributes to describe a single rover.

_check_field_name_clashes()
Ref #17673.

_check_fields(***kwargs*)
Perform all field checks.

_check_id_field()
Check if *id* field is a primary key.

_check_index_together()
Check the value of "index_together" option.

_check_long_column_names()
Check that any auto-generated column names are shorter than the limits for each database in which the model will be created.

_check_m2m_through_same_relationship()
Check if no relationship model is used by more than one m2m field.

_check_managers(***kwargs*)
Perform all manager checks.

_check_ordering()
Check "ordering" option – is it a list of strings and do all fields exist?

_check_swappable()
Check if the swapped model exists.

_check_unique_together()
Check the value of "unique_together" option.

_do_insert(*manager*, *using*, *fields*, *update_pk*, *raw*)
Do an INSERT. If update_pk is defined then this method should return the new pk for the model.

_do_update(*base_qs*, *using*, *pk_val*, *values*, *update_fields*, *forced_update*)
This method will try to update the model. If the model was updated (in the sense that an update query was done and a matching row was found from the DB) the method will return True.

_get_unique_checks(*exclude=None*)
Gather a list of checks to perform. Since validate_unique could be called from a ModelForm, some fields may have been excluded; we can't perform a unique check on a model that is missing fields involved in that check. Fields that did not validate should also be excluded, but they need to be passed in via the exclude argument.

_save_parents(*cls*, *using*, *update_fields*)
Saves all the parents of cls using values from self.

_save_table(*raw=False*, *cls=None*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Does the heavy-lifting involved in saving. Updates or inserts the data for a single table.

clean()
Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

clean_fields(*exclude=None*)
Cleans all fields and raises a ValidationError containing a dict of all validation errors if any occur.

full_clean(*exclude=None*, *validate_unique=True*)
Calls clean_fields, clean, and validate_unique, on the model, and raises a `ValidationError` for any errors that occurred.

get_deferred_fields()
Returns a set containing names of deferred fields for this instance.

refresh_from_db(*using=None*, *fields=None*)
Reloads field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

save(*force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Handles the parts of saving which should be done only once per save, yet need to be done in raw saves, too. This includes some sanity checks and signal sending.

The 'raw' argument is telling save_base not to save any parent models and not to do any changes to the values before save. This is used by fixture loading.

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

validate_unique(*exclude=None*)
Checks unique constraints on the model and raises `ValidationError` if any failed.

mission_control.serializers

Mission Control serializers.

Classes

`BlockDiagram`(*args, **kwargs)	Attributes to describe a single block diagram.
`BlockDiagramSerializer`([instance, data])	Block diagram model serializer.
`Rover`(*args, **kwargs)	Attributes to describe a single rover.
`RoverSerializer`([instance, data])	Rover model serializer.

class mission_control.serializers.**BlockDiagramSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Bases: rest_framework.serializers.ModelSerializer

Block diagram model serializer.

class Meta

Meta class.

model

alias of BlockDiagram

BlockDiagramSerializer.**_get_model_fields**(*field_names*, *declared_fields*, *extra_kwargs*)

Returns all the model fields that are being mapped to by fields on the serializer class. Returned as a dict of 'model field name' -> 'model field'. Used internally by *get_uniqueness_field_options*.

BlockDiagramSerializer.**bind**(*field_name*, *parent*)

Initializes the field name and parent for the field instance. Called when a field is added to the parent serializer instance.

BlockDiagramSerializer.**build_field**(*field_name*, *info*, *model_class*, *nested_depth*)

Return a two tuple of (cls, kwargs) to build a serializer field with.

BlockDiagramSerializer.**build_nested_field**(*field_name*, *relation_info*, *nested_depth*)

Create nested fields for forward and reverse relationships.

BlockDiagramSerializer.**build_property_field**(*field_name*, *model_class*)

Create a read only field for model methods and properties.

BlockDiagramSerializer.**build_relational_field**(*field_name*, *relation_info*)

Create fields for forward and reverse relationships.

BlockDiagramSerializer.**build_standard_field**(*field_name*, *model_field*)

Create regular model fields.

BlockDiagramSerializer.**build_unknown_field**(*field_name*, *model_class*)

Raise an error on any unknown fields.

BlockDiagramSerializer.**build_url_field**(*field_name*, *model_class*)

Create a field representing the object's own URL.

BlockDiagramSerializer.**context**

Returns the context as passed to the root serializer on initialization.

BlockDiagramSerializer.**create**(*validated_data*)

We have a bit of extra checking around this in order to provide descriptive messages when something goes wrong, but this method is essentially just:

return ExampleModel.objects.create(**validated_data)

If there are many to many fields present on the instance then they cannot be set until the model is instantiated, in which case the implementation is like so:

example_relationship = validated_data.pop('example_relationship') instance = ExampleModel.objects.create(**validated_data) instance.example_relationship = example_relationship return instance

The default implementation also does not handle nested relationships. If you want to support writable nested relationships you'll need to write an explicit *.create()* method.

BlockDiagramSerializer.**default_empty_html**
alias of empty

BlockDiagramSerializer.**fail**(*key*, ***kwargs*)
A helper method that simply raises a validation error.

BlockDiagramSerializer.**fields**
A dictionary of {field_name: field_instance}.

BlockDiagramSerializer.**get_attribute**(*instance*)
Given the *outgoing* object instance, return the primitive value that should be used for this field.

BlockDiagramSerializer.**get_default**()
Return the default value to use when validating data if no input is provided for this field.

If a default has not been set for this field then this will simply raise *SkipField*, indicating that no value should be set in the validated data for this field.

BlockDiagramSerializer.**get_default_field_names**(*declared_fields*, *model_info*)
Return the default list of field names that will be used if the *Meta.fields* option is not specified.

BlockDiagramSerializer.**get_extra_kwargs**()
Return a dictionary mapping field names to a dictionary of additional keyword arguments.

BlockDiagramSerializer.**get_field_names**(*declared_fields*, *info*)
Returns the list of all field names that should be created when instantiating this serializer class. This is based on the default set of fields, but also takes into account the *Meta.fields* or *Meta.exclude* options if they have been specified.

BlockDiagramSerializer.**get_fields**()
Return the dict of field names -> field instances that should be used for *self.fields* when instantiating the serializer.

BlockDiagramSerializer.**get_unique_for_date_validators**()
Determine a default set of validators for the following constraints:

- unique_for_date
- unique_for_month
- unique_for_year

BlockDiagramSerializer.**get_unique_together_validators**()
Determine a default set of validators for any unique_together constraints.

BlockDiagramSerializer.**get_uniqueness_extra_kwargs**(*field_names*, *declared_fields*, *extra_kwargs*)
Return any additional field options that need to be included as a result of uniqueness constraints on the model. This is returned as a two-tuple of:

('dict of updated extra kwargs', 'mapping of hidden fields')

BlockDiagramSerializer.**get_validators**()
Determine the set of validators to use when instantiating serializer.

BlockDiagramSerializer.**include_extra_kwargs**(*kwargs*, *extra_kwargs*)
Include any 'extra_kwargs' that have been included for this field, possibly removing any incompatible existing keyword arguments.

BlockDiagramSerializer.**many_init**(**args*, ***kwargs*)
This method implements the creation of a *ListSerializer* parent class when *many=True* is used. You can customize it if you need to control which keyword arguments are passed to the parent, and which are passed to the child.

Note that we're over-cautious in passing most arguments to both parent and child classes in order to try to cover the general case. If you're overriding this method you'll probably want something much simpler, eg:

@classmethod def many_init(cls, *args, **kwargs):

kwargs['child'] = cls() return CustomListSerializer(*args, **kwargs)

`BlockDiagramSerializer.`**`root`**
Returns the top-level serializer for this field.

`BlockDiagramSerializer.`**`run_validation`**(*data=<class rest_framework.fields.empty>*)
We override the default *run_validation*, because the validation performed by validators and the *.validate()* method should be coerced into an error dictionary with a 'non_fields_error' key.

`BlockDiagramSerializer.`**`run_validators`**(*value*)
Test the given value against all the validators on the field, and either raise a *ValidationError* or simply return.

`BlockDiagramSerializer.`**`serializer_related_to_field`**
alias of `SlugRelatedField`

`BlockDiagramSerializer.`**`serializer_url_field`**
alias of `HyperlinkedIdentityField`

`BlockDiagramSerializer.`**`to_internal_value`**(*data*)
Dict of native values <- Dict of primitive datatypes.

`BlockDiagramSerializer.`**`to_representation`**(*instance*)
Object instance -> Dict of primitive datatypes.

`BlockDiagramSerializer.`**`validate_empty_values`**(*data*)
Validate empty values, and either:

- Raise *ValidationError*, indicating invalid data.
- Raise *SkipField*, indicating that the field should be ignored.
- Return (True, data), indicating an empty value that should be returned without any further validation being applied.
- Return (False, data), indicating a non-empty value, that should have validation applied as normal.

class `mission_control.serializers.`**`RoverSerializer`**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)
Bases: `rest_framework.serializers.HyperlinkedModelSerializer`

Rover model serializer.

class **`Meta`**
Meta class.

`model`
alias of `Rover`

`RoverSerializer.`**`_get_model_fields`**(*field_names*, *declared_fields*, *extra_kwargs*)
Returns all the model fields that are being mapped to by fields on the serializer class. Returned as a dict of 'model field name' -> 'model field'. Used internally by *get_uniqueness_field_options*.

`RoverSerializer.`**`bind`**(*field_name*, *parent*)
Initializes the field name and parent for the field instance. Called when a field is added to the parent serializer instance.

`RoverSerializer.`**`build_field`**(*field_name*, *info*, *model_class*, *nested_depth*)
Return a two tuple of (cls, kwargs) to build a serializer field with.

RoverSerializer.**build_nested_field**(*field_name*, *relation_info*, *nested_depth*)
Create nested fields for forward and reverse relationships.

RoverSerializer.**build_property_field**(*field_name*, *model_class*)
Create a read only field for model methods and properties.

RoverSerializer.**build_relational_field**(*field_name*, *relation_info*)
Create fields for forward and reverse relationships.

RoverSerializer.**build_standard_field**(*field_name*, *model_field*)
Create regular model fields.

RoverSerializer.**build_unknown_field**(*field_name*, *model_class*)
Raise an error on any unknown fields.

RoverSerializer.**build_url_field**(*field_name*, *model_class*)
Create a field representing the object's own URL.

RoverSerializer.**context**
Returns the context as passed to the root serializer on initialization.

RoverSerializer.**create**(*validated_data*)
We have a bit of extra checking around this in order to provide descriptive messages when something goes wrong, but this method is essentially just:

return ExampleModel.objects.create(**validated_data)

If there are many to many fields present on the instance then they cannot be set until the model is instantiated, in which case the implementation is like so:

example_relationship = validated_data.pop('example_relationship') instance = ExampleModel.objects.create(**validated_data) instance.example_relationship = example_relationship return instance

The default implementation also does not handle nested relationships. If you want to support writable nested relationships you'll need to write an explicit *.create()* method.

RoverSerializer.**default_empty_html**
alias of `empty`

RoverSerializer.**fail**(*key*, ***kwargs*)
A helper method that simply raises a validation error.

RoverSerializer.**fields**
A dictionary of {field_name: field_instance}.

RoverSerializer.**get_attribute**(*instance*)
Given the *outgoing* object instance, return the primitive value that should be used for this field.

RoverSerializer.**get_default**()
Return the default value to use when validating data if no input is provided for this field.

If a default has not been set for this field then this will simply raise *SkipField*, indicating that no value should be set in the validated data for this field.

RoverSerializer.**get_default_field_names**(*declared_fields*, *model_info*)
Return the default list of field names that will be used if the *Meta.fields* option is not specified.

RoverSerializer.**get_extra_kwargs**()
Return a dictionary mapping field names to a dictionary of additional keyword arguments.

RoverSerializer.**get_field_names**(*declared_fields*, *info*)
Returns the list of all field names that should be created when instantiating this serializer class. This is

based on the default set of fields, but also takes into account the *Meta.fields* or *Meta.exclude* options if they have been specified.

RoverSerializer.**get_fields**()
Return the dict of field names -> field instances that should be used for *self.fields* when instantiating the serializer.

RoverSerializer.**get_unique_for_date_validators**()
Determine a default set of validators for the following constraints:

•unique_for_date

•unique_for_month

•unique_for_year

RoverSerializer.**get_unique_together_validators**()
Determine a default set of validators for any unique_together constraints.

RoverSerializer.**get_uniqueness_extra_kwargs**(*field_names*, *declared_fields*, *extra_kwargs*)
Return any additional field options that need to be included as a result of uniqueness constraints on the model. This is returned as a two-tuple of:

('dict of updated extra kwargs', 'mapping of hidden fields')

RoverSerializer.**get_validators**()
Determine the set of validators to use when instantiating serializer.

RoverSerializer.**include_extra_kwargs**(*kwargs*, *extra_kwargs*)
Include any 'extra_kwargs' that have been included for this field, possibly removing any incompatible existing keyword arguments.

RoverSerializer.**many_init**(**args*, ***kwargs*)
This method implements the creation of a *ListSerializer* parent class when *many=True* is used. You can customize it if you need to control which keyword arguments are passed to the parent, and which are passed to the child.

Note that we're over-cautious in passing most arguments to both parent and child classes in order to try to cover the general case. If you're overriding this method you'll probably want something much simpler, eg:

@classmethod def many_init(cls, *args, **kwargs):

kwargs['child'] = cls() return CustomListSerializer(*args, **kwargs)

RoverSerializer.**root**
Returns the top-level serializer for this field.

RoverSerializer.**run_validation**(*data=<class rest_framework.fields.empty>*)
We override the default *run_validation*, because the validation performed by validators and the *.validate()* method should be coerced into an error dictionary with a 'non_fields_error' key.

RoverSerializer.**run_validators**(*value*)
Test the given value against all the validators on the field, and either raise a *ValidationError* or simply return.

RoverSerializer.**serializer_related_to_field**
alias of SlugRelatedField

RoverSerializer.**serializer_url_field**
alias of HyperlinkedIdentityField

RoverSerializer.**to_internal_value**(*data*)
Dict of native values <- Dict of primitive datatypes.

RoverSerializer.**to_representation**(*instance*)
Object instance -> Dict of primitive datatypes.

RoverSerializer.**validate_empty_values**(*data*)
Validate empty values, and either:

- Raise *ValidationError*, indicating invalid data.
- Raise *SkipField*, indicating that the field should be ignored.
- Return (True, data), indicating an empty value that should be returned without any further validation being applied.
- Return (False, data), indicating a non-empty value, that should have validation applied as normal.

rovercode-web

This is the main website. It handles all of the non-Mission-Control things, like serving the landing page and logging in users.

config.urls
rovercode_web.users.adapters
rovercode_web.users.admin
rovercode_web.users.apps
rovercode_web.users.models
rovercode_web.users.urls
rovercode_web.users.views

rovercode_web.users.adapters

Classes

AccountAdapter([request])	
DefaultAccountAdapter([request])	
DefaultSocialAccountAdapter([request])	
SocialAccountAdapter([request])	

rovercode_web.users.admin

Classes

AuthUserAdmin	alias of UserAdmin
MyUserAdmin(model, admin_site)	
MyUserChangeForm(*args, **kwargs)	
MyUserCreationForm(*args, **kwargs)	
User(id, password, last_login, is_superuser, ...)	
UserChangeForm(*args, **kwargs)	
UserCreationForm(*args, **kwargs)	A form that creates a user, with no privileges, from the given username and password.

rovercode_web.users.apps

Classes

AppConfig(app_name, app_module)	Class representing a Django application and its configuration.
UsersConfig(app_name, app_module)	

rovercode_web.users.models

Functions

python_2_unicode_compatible(klass)	A decorator that defines __unicode__ and __str__ methods under Python 2.
reverse(viewname[, urlconf, args, kwargs, ...])	

Classes

AbstractUser(*args, **kwargs)	An abstract base class implementing a fully featured User model with admin-compliant permissions.
User(id, password, last_login, is_superuser, ...)	

class rovercode_web.users.models.**User**(*id*, *password*, *last_login*, *is_superuser*, *username*, *first_name*, *last_name*, *email*, *is_staff*, *is_active*, *date_joined*, *name*)

Bases: django.contrib.auth.models.AbstractUser

_check_field_name_clashes()
Ref #17673.

_check_fields(***kwargs*)
Perform all field checks.

_check_id_field()
Check if *id* field is a primary key.

_check_index_together()
Check the value of "index_together" option.

_check_long_column_names()
Check that any auto-generated column names are shorter than the limits for each database in which the model will be created.

_check_m2m_through_same_relationship()
Check if no relationship model is used by more than one m2m field.

_check_managers(***kwargs*)
Perform all manager checks.

_check_ordering()
Check "ordering" option – is it a list of strings and do all fields exist?

_check_swappable()
Check if the swapped model exists.

_check_unique_together()
Check the value of "unique_together" option.

_do_insert(*manager*, *using*, *fields*, *update_pk*, *raw*)
Do an INSERT. If update_pk is defined then this method should return the new pk for the model.

_do_update(*base_qs*, *using*, *pk_val*, *values*, *update_fields*, *forced_update*)
This method will try to update the model. If the model was updated (in the sense that an update query was done and a matching row was found from the DB) the method will return True.

_get_unique_checks(*exclude=None*)
Gather a list of checks to perform. Since validate_unique could be called from a ModelForm, some fields may have been excluded; we can't perform a unique check on a model that is missing fields involved in that check. Fields that did not validate should also be excluded, but they need to be passed in via the exclude argument.

_save_parents(*cls*, *using*, *update_fields*)
Saves all the parents of cls using values from self.

_save_table(*raw=False*, *cls=None*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Does the heavy-lifting involved in saving. Updates or inserts the data for a single table.

check_password(*raw_password*)
Return a boolean of whether the raw_password was correct. Handles hashing formats behind the scenes.

clean_fields(*exclude=None*)
Cleans all fields and raises a ValidationError containing a dict of all validation errors if any occur.

email_user(*subject*, *message*, *from_email=None*, ***kwargs*)
Sends an email to this User.

full_clean(*exclude=None*, *validate_unique=True*)
Calls clean_fields, clean, and validate_unique, on the model, and raises a `ValidationError` for any errors that occurred.

get_deferred_fields()
Returns a set containing names of deferred fields for this instance.

get_full_name()
Returns the first_name plus the last_name, with a space in between.

get_group_permissions(*obj=None*)
Returns a list of permission strings that this user has through their groups. This method queries all available auth backends. If an object is passed in, only permissions matching this object are returned.

get_session_auth_hash()
Return an HMAC of the password field.

get_short_name()
Returns the short name for the user.

get_username()
Return the identifying username for this User

has_module_perms(*app_label*)
Returns True if the user has any permissions in the given app label. Uses pretty much the same logic as has_perm, above.

has_perm(*perm*, *obj=None*)
Returns True if the user has the specified permission. This method queries all available auth backends, but returns immediately if any backend returns True. Thus, a user who has permission from a single auth backend is assumed to have permission in general. If an object is provided, permissions for this specific object are checked.

has_perms(*perm_list*, *obj=None*)
Returns True if the user has each of the specified permissions. If object is passed, it checks if the user has all required perms for this object.

is_anonymous
Always return False. This is a way of comparing User objects to anonymous users.

is_authenticated
Always return True. This is a way to tell if the user has been authenticated in templates.

refresh_from_db(*using=None*, *fields=None*)
Reloads field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

save_base(*raw=False*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)
Handles the parts of saving which should be done only once per save, yet need to be done in raw saves, too. This includes some sanity checks and signal sending.

The 'raw' argument is telling save_base not to save any parent models and not to do any changes to the values before save. This is used by fixture loading.

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

validate_unique(*exclude=None*)
Checks unique constraints on the model and raises `ValidationError` if any failed.

rovercode_web.users.urls

Functions

`url`(regex, view[, kwargs, name])

rovercode_web.users.views

Functions

`reverse`(viewname[, urlconf, args, kwargs, ...])

Classes

DetailView(**kwargs)	Render a "detail" view of an object.
ListView(**kwargs)	Render some list of objects, set by *self.model* or *self.queryset*.
LoginRequiredMixin	CBV mixin which verifies that the current user is authenticated.
RedirectView(**kwargs)	A view that provides a redirect on any GET request.
UpdateView(**kwargs)	View for updating an object, with a response rendered by template.
User(id, password, last_login, is_superuser, ...)	
UserDetailView(**kwargs)	Constructor.
UserListView(**kwargs)	Constructor.
UserRedirectView(**kwargs)	Constructor.
UserUpdateView(**kwargs)	Constructor.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

r

Index

Symbols

B

C

D

I

M

R

S

T

U

V