# Agent

should remember defaults

if (typeof Promise === 'undefined') {  
 return;  
}  
let called = 0;  
let event\_called = 0;  
const agent = request  
 .agent()  
 .accept('json')  
 .use(() => {  
 called++;  
 })  
 .once('request', () => {  
 event\_called++;  
 })  
 .query({ hello: 'world' })  
 .set('X-test', 'testing');  
assert.equal(0, called);  
assert.equal(0, event\_called);  
return agent  
 .get(`${base}/echo`)  
 .then(res => {  
 assert.equal(1, called);  
 assert.equal(1, event\_called);  
 assert.equal('application/json', res.headers.accept);  
 assert.equal('testing', res.headers['x-test']);  
 return agent.get(`${base}/querystring`);  
 })  
 .then(res => {  
 assert.equal(2, called);  
 assert.equal(2, event\_called);  
 assert.deepEqual({ hello: 'world' }, res.body);  
 });

# request

# res.statusCode

should set statusCode

done => {  
 request.get(`${uri}/login`, (err, res) => {  
 try {  
 assert.strictEqual(res.statusCode, 200);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# should allow the send shorthand

with callback in the method call

done => {  
 request.get(`${uri}/login`, (err, res) => {  
 assert.equal(res.status, 200);  
 done();  
 });  
 }

with data in the method call

done => {  
 request.post(`${uri}/echo`, { foo: 'bar' }).end((err, res) => {  
 assert.equal('{"foo":"bar"}', res.text);  
 done();  
 });  
 }

with callback and data in the method call

done => {  
 request.post(`${uri}/echo`, { foo: 'bar' }, (err, res) => {  
 assert.equal('{"foo":"bar"}', res.text);  
 done();  
 });  
 }

# with a callback

should invoke .end()

done => {  
 request.get(`${uri}/login`, (err, res) => {  
 try {  
 assert.equal(res.status, 200);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# .end()

should issue a request

done => {  
 request.get(`${uri}/login`).end((err, res) => {  
 try {  
 assert.equal(res.status, 200);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

is optional with a promise

if (typeof Promise === 'undefined') {  
 return;  
}  
return request  
 .get(`${uri}/login`)  
 .then(res => res.status)  
 .then()  
 .then(status => {  
 assert.equal(200, status, 'Real promises pass results through');  
 });

called only once with a promise

if (typeof Promise === 'undefined') {  
 return;  
}  
const req = request.get(`${uri}/unique`);  
return Promise.all([req, req, req]).then(results => {  
 results.forEach(item => {  
 assert.equal(  
 item.body,  
 results[0].body,  
 'It should keep returning the same result after being called once'  
 );  
 });  
});

# res.error

ok

done => {  
 let calledErrorEvent = false;  
 let calledOKHandler = false;  
 request  
 .get(`${uri}/error`)  
 .ok(res => {  
 assert.strictEqual(500, res.status);  
 calledOKHandler = true;  
 return true;  
 })  
 .on('error', err => {  
 calledErrorEvent = true;  
 })  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert.strictEqual(res.status, 500);  
 assert(!calledErrorEvent);  
 assert(calledOKHandler);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should should be an Error object

done => {  
 let calledErrorEvent = false;  
 request  
 .get(`${uri}/error`)  
 .on('error', err => {  
 assert.strictEqual(err.status, 500);  
 calledErrorEvent = true;  
 })  
 .end((err, res) => {  
 try {  
 if (NODE) {  
 res.error.message.should.equal('cannot GET /error (500)');  
 } else {  
 res.error.message.should.equal(`cannot GET ${uri}/error (500)`);  
 }  
 assert.strictEqual(res.error.status, 500);  
 assert(err, 'should have an error for 500');  
 assert.equal(err.message, 'Internal Server Error');  
 assert(calledErrorEvent);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

with .then() promise

if (typeof Promise === 'undefined') {  
 return;  
}  
return request.get(`${uri}/error`).then(  
 () => {  
 assert.fail();  
 },  
 err => {  
 assert.equal(err.message, 'Internal Server Error');  
 }  
);

with .ok() returning false

if (typeof Promise === 'undefined') {  
 return;  
}  
return request  
 .get(`${uri}/echo`)  
 .ok(() => false)  
 .then(  
 () => {  
 assert.fail();  
 },  
 err => {  
 assert.equal(200, err.response.status);  
 assert.equal(err.message, 'OK');  
 }  
 );

with .ok() throwing an Error

if (typeof Promise === 'undefined') {  
 return;  
}  
return request  
 .get(`${uri}/echo`)  
 .ok(() => {  
 throw new Error('boom');  
 })  
 .then(  
 () => {  
 assert.fail();  
 },  
 err => {  
 assert.equal(200, err.response.status);  
 assert.equal(err.message, 'boom');  
 }  
 );

# res.header

should be an object

done => {  
 request.get(`${uri}/login`).end((err, res) => {  
 try {  
 assert.equal('Express', res.header['x-powered-by']);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# set headers

should only set headers for ownProperties of header

done => {  
 try {  
 request  
 .get(`${uri}/echo-headers`)  
 .set('valid', 'ok')  
 .end((err, res) => {  
 if (  
 !err &&  
 res.body &&  
 res.body.valid &&  
 !res.body.hasOwnProperty('invalid')  
 ) {  
 return done();  
 }  
 done(err || new Error('fail'));  
 });  
 } catch (err) {  
 done(err);  
 }  
 }

# res.charset

should be set when present

done => {  
 request.get(`${uri}/login`).end((err, res) => {  
 try {  
 res.charset.should.equal('utf-8');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# res.statusType

should provide the first digit

done => {  
 request.get(`${uri}/login`).end((err, res) => {  
 try {  
 assert(!err, 'should not have an error for success responses');  
 assert.equal(200, res.status);  
 assert.equal(2, res.statusType);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# res.type

should provide the mime-type void of params

done => {  
 request.get(`${uri}/login`).end((err, res) => {  
 try {  
 res.type.should.equal('text/html');  
 res.charset.should.equal('utf-8');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.set(field, val)

should set the header field

done => {  
 request  
 .post(`${uri}/echo`)  
 .set('X-Foo', 'bar')  
 .set('X-Bar', 'baz')  
 .end((err, res) => {  
 try {  
 assert.equal('bar', res.header['x-foo']);  
 assert.equal('baz', res.header['x-bar']);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.set(obj)

should set the header fields

done => {  
 request  
 .post(`${uri}/echo`)  
 .set({ 'X-Foo': 'bar', 'X-Bar': 'baz' })  
 .end((err, res) => {  
 try {  
 assert.equal('bar', res.header['x-foo']);  
 assert.equal('baz', res.header['x-bar']);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.type(str)

should set the Content-Type

done => {  
 request  
 .post(`${uri}/echo`)  
 .type('text/x-foo')  
 .end((err, res) => {  
 try {  
 res.header['content-type'].should.equal('text/x-foo');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should map "json"

done => {  
 request  
 .post(`${uri}/echo`)  
 .type('json')  
 .send('{"a": 1}')  
 .end((err, res) => {  
 try {  
 res.should.be.json();  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should map "html"

done => {  
 request  
 .post(`${uri}/echo`)  
 .type('html')  
 .end((err, res) => {  
 try {  
 res.header['content-type'].should.equal('text/html');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.accept(str)

should set Accept

done => {  
 request  
 .get(`${uri}/echo`)  
 .accept('text/x-foo')  
 .end((err, res) => {  
 try {  
 res.header.accept.should.equal('text/x-foo');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should map "json"

done => {  
 request  
 .get(`${uri}/echo`)  
 .accept('json')  
 .end((err, res) => {  
 try {  
 res.header.accept.should.equal('application/json');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should map "xml"

done => {  
 request  
 .get(`${uri}/echo`)  
 .accept('xml')  
 .end((err, res) => {  
 try {  
 // Mime module keeps changing this :(  
 assert(  
 res.header.accept == 'application/xml' ||  
 res.header.accept == 'text/xml'  
 );  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should map "html"

done => {  
 request  
 .get(`${uri}/echo`)  
 .accept('html')  
 .end((err, res) => {  
 try {  
 res.header.accept.should.equal('text/html');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.send(str)

should write the string

done => {  
 request  
 .post(`${uri}/echo`)  
 .type('json')  
 .send('{"name":"tobi"}')  
 .end((err, res) => {  
 try {  
 res.text.should.equal('{"name":"tobi"}');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.send(Object)

should default to json

done => {  
 request  
 .post(`${uri}/echo`)  
 .send({ name: 'tobi' })  
 .end((err, res) => {  
 try {  
 res.should.be.json();  
 res.text.should.equal('{"name":"tobi"}');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# when called several times

should merge the objects

done => {  
 request  
 .post(`${uri}/echo`)  
 .send({ name: 'tobi' })  
 .send({ age: 1 })  
 .end((err, res) => {  
 try {  
 res.should.be.json();  
 if (NODE) {  
 res.buffered.should.be.true();  
 }  
 res.text.should.equal('{"name":"tobi","age":1}');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# .end(fn)

should check arity

done => {  
 request  
 .post(`${uri}/echo`)  
 .send({ name: 'tobi' })  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 res.text.should.equal('{"name":"tobi"}');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should emit request

done => {  
 const req = request.post(`${uri}/echo`);  
 req.on('request', request => {  
 assert.equal(req, request);  
 done();  
 });  
 req.end();  
 }

should emit response

done => {  
 request  
 .post(`${uri}/echo`)  
 .send({ name: 'tobi' })  
 .on('response', res => {  
 res.text.should.equal('{"name":"tobi"}');  
 done();  
 })  
 .end();  
 }

# .then(fulfill, reject)

should support successful fulfills with .then(fulfill)

done => {  
 if (typeof Promise === 'undefined') {  
 return done();  
 }  
 request  
 .post(`${uri}/echo`)  
 .send({ name: 'tobi' })  
 .then(res => {  
 res.type.should.equal('application/json');  
 res.text.should.equal('{"name":"tobi"}');  
 done();  
 });  
 }

should reject an error with .then(null, reject)

done => {  
 if (typeof Promise === 'undefined') {  
 return done();  
 }  
 request.get(`${uri}/error`).then(null, err => {  
 assert.equal(err.status, 500);  
 assert.equal(err.response.text, 'boom');  
 done();  
 });  
 }

# .catch(reject)

should reject an error with .catch(reject)

done => {  
 if (typeof Promise === 'undefined') {  
 return done();  
 }  
 request.get(`${uri}/error`).catch(err => {  
 assert.equal(err.status, 500);  
 assert.equal(err.response.text, 'boom');  
 done();  
 });  
 }

# .abort()

should abort the request

done => {  
 const req = request.get(`${uri}/delay/3000`);  
 req.end((err, res) => {  
 try {  
 assert(false, 'should not complete the request');  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 req.on('error', error => {  
 done(error);  
 });  
 req.on('abort', done);  
 setTimeout(() => {  
 req.abort();  
 }, 500);  
 }

should abort the promise

const req = request.get(`${uri}/delay/3000`);  
setTimeout(() => {  
 req.abort();  
}, 10);  
return req.then(  
 () => {  
 assert.fail('should not complete the request');  
 },  
 err => {  
 assert.equal('ABORTED', err.code);  
 }  
);

should allow chaining .abort() several times

done => {  
 const req = request.get(`${uri}/delay/3000`);  
 req.end((err, res) => {  
 try {  
 assert(false, 'should not complete the request');  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 // This also verifies only a single 'done' event is emitted  
 req.on('abort', done);  
 setTimeout(() => {  
 req  
 .abort()  
 .abort()  
 .abort();  
 }, 1000);  
 }

should not allow abort then end

done => {  
 request  
 .get(`${uri}/delay/3000`)  
 .abort()  
 .end((err, res) => {  
 done(err ? undefined : new Error('Expected abort error'));  
 });  
 }

# req.toJSON()

should describe the request

done => {  
 const req = request.post(`${uri}/echo`).send({ foo: 'baz' });  
 req.end((err, res) => {  
 try {  
 const json = req.toJSON();  
 assert.equal('POST', json.method);  
 assert(/\/echo$/.test(json.url));  
 assert.equal('baz', json.data.foo);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.options()

should allow request body

done => {  
 request  
 .options(`${uri}/options/echo/body`)  
 .send({ foo: 'baz' })  
 .end((err, res) => {  
 try {  
 assert.equal(err, null);  
 assert.strictEqual(res.body.foo, 'baz');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.sortQuery()

nop with no querystring

done => {  
 request  
 .get(`${uri}/url`)  
 .sortQuery()  
 .end((err, res) => {  
 try {  
 assert.equal(res.text, '/url');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should sort the request querystring

done => {  
 request  
 .get(`${uri}/url`)  
 .query('search=Manny')  
 .query('order=desc')  
 .sortQuery()  
 .end((err, res) => {  
 try {  
 assert.equal(res.text, '/url?order=desc&search=Manny');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should allow disabling sorting

done => {  
 request  
 .get(`${uri}/url`)  
 .query('search=Manny')  
 .query('order=desc')  
 .sortQuery() // take default of true  
 .sortQuery(false) // override it in later call  
 .end((err, res) => {  
 try {  
 assert.equal(res.text, '/url?search=Manny&order=desc');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should sort the request querystring using customized function

done => {  
 request  
 .get(`${uri}/url`)  
 .query('name=Nick')  
 .query('search=Manny')  
 .query('order=desc')  
 .sortQuery((a, b) => a.length - b.length)  
 .end((err, res) => {  
 try {  
 assert.equal(res.text, '/url?name=Nick&order=desc&search=Manny');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.set("Content-Type", contentType)

should work with just the contentType component

done => {  
 request  
 .post(`${uri}/echo`)  
 .set('Content-Type', 'application/json')  
 .send({ name: 'tobi' })  
 .end((err, res) => {  
 assert(!err);  
 done();  
 });  
 }

should work with the charset component

done => {  
 request  
 .post(`${uri}/echo`)  
 .set('Content-Type', 'application/json; charset=utf-8')  
 .send({ name: 'tobi' })  
 .end((err, res) => {  
 assert(!err);  
 done();  
 });  
 }

# req.send(Object) as "form"

# with req.type() set to form

should send x-www-form-urlencoded data

done => {  
 request  
 .post(`${base}/echo`)  
 .type('form')  
 .send({ name: 'tobi' })  
 .end((err, res) => {  
 res.header['content-type'].should.equal(  
 'application/x-www-form-urlencoded'  
 );  
 res.text.should.equal('name=tobi');  
 done();  
 });  
 }

# when called several times

should merge the objects

done => {  
 request  
 .post(`${base}/echo`)  
 .type('form')  
 .send({ name: { first: 'tobi', last: 'holowaychuk' } })  
 .send({ age: '1' })  
 .end((err, res) => {  
 res.header['content-type'].should.equal(  
 'application/x-www-form-urlencoded'  
 );  
 res.text.should.equal(  
 'name%5Bfirst%5D=tobi&name%5Blast%5D=holowaychuk&age=1'  
 );  
 done();  
 });  
 }

# req.attach

ignores null file

done => {  
 request  
 .post('/echo')  
 .attach('image', null)  
 .end((err, res) => {  
 done();  
 });  
 }

# req.field

allow bools

done => {  
 if (!formDataSupported) {  
 return done();  
 }  
 request  
 .post(`${base}/formecho`)  
 .field('bools', true)  
 .field('strings', 'true')  
 .end((err, res) => {  
 assert.ifError(err);  
 assert.deepStrictEqual(res.body, { bools: 'true', strings: 'true' });  
 done();  
 });  
 }

allow objects

done => {  
 if (!formDataSupported) {  
 return done();  
 }  
 request  
 .post(`${base}/formecho`)  
 .field({ bools: true, strings: 'true' })  
 .end((err, res) => {  
 assert.ifError(err);  
 assert.deepStrictEqual(res.body, { bools: 'true', strings: 'true' });  
 done();  
 });  
 }

works with arrays in objects

done => {  
 if (!formDataSupported) {  
 return done();  
 }  
 request  
 .post(`${base}/formecho`)  
 .field({ numbers: [1, 2, 3] })  
 .end((err, res) => {  
 assert.ifError(err);  
 assert.deepStrictEqual(res.body, { numbers: ['1', '2', '3'] });  
 done();  
 });  
 }

works with arrays

done => {  
 if (!formDataSupported) {  
 return done();  
 }  
 request  
 .post(`${base}/formecho`)  
 .field('letters', ['a', 'b', 'c'])  
 .end((err, res) => {  
 assert.ifError(err);  
 assert.deepStrictEqual(res.body, { letters: ['a', 'b', 'c'] });  
 done();  
 });  
 }

throw when empty

should.throws(() => {  
 request.post(`${base}/echo`).field();  
}, /name/);  
should.throws(() => {  
 request.post(`${base}/echo`).field('name');  
}, /val/);

cannot be mixed with send()

assert.throws(() => {  
 request  
 .post('/echo')  
 .field('form', 'data')  
 .send('hi');  
});  
assert.throws(() => {  
 request  
 .post('/echo')  
 .send('hi')  
 .field('form', 'data');  
});

# req.send(Object) as "json"

should default to json

done => {  
 request  
 .post(`${uri}/echo`)  
 .send({ name: 'tobi' })  
 .end((err, res) => {  
 res.should.be.json();  
 res.text.should.equal('{"name":"tobi"}');  
 done();  
 });  
 }

should work with arrays

done => {  
 request  
 .post(`${uri}/echo`)  
 .send([1, 2, 3])  
 .end((err, res) => {  
 res.should.be.json();  
 res.text.should.equal('[1,2,3]');  
 done();  
 });  
 }

should work with value null

done => {  
 request  
 .post(`${uri}/echo`)  
 .type('json')  
 .send('null')  
 .end((err, res) => {  
 res.should.be.json();  
 assert.strictEqual(res.body, null);  
 done();  
 });  
 }

should work with value false

done => {  
 request  
 .post(`${uri}/echo`)  
 .type('json')  
 .send('false')  
 .end((err, res) => {  
 res.should.be.json();  
 res.body.should.equal(false);  
 done();  
 });  
 }

should work with value 0

done => {  
 // fails in IE9  
 request  
 .post(`${uri}/echo`)  
 .type('json')  
 .send('0')  
 .end((err, res) => {  
 res.should.be.json();  
 res.body.should.equal(0);  
 done();  
 });  
 }

should work with empty string value

done => {  
 request  
 .post(`${uri}/echo`)  
 .type('json')  
 .send('""')  
 .end((err, res) => {  
 res.should.be.json();  
 res.body.should.equal('');  
 done();  
 });  
 }

should work with GET

done => {  
 request  
 .get(`${uri}/echo`)  
 .send({ tobi: 'ferret' })  
 .end((err, res) => {  
 try {  
 res.should.be.json();  
 res.text.should.equal('{"tobi":"ferret"}');  
 ({ tobi: 'ferret' }.should.eql(res.body));  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should work with vendor MIME type

done => {  
 request  
 .post(`${uri}/echo`)  
 .set('Content-Type', 'application/vnd.example+json')  
 .send({ name: 'vendor' })  
 .end((err, res) => {  
 res.text.should.equal('{"name":"vendor"}');  
 ({ name: 'vendor' }.should.eql(res.body));  
 done();  
 });  
 }

# when called several times

should merge the objects

done => {  
 request  
 .post(`${uri}/echo`)  
 .send({ name: 'tobi' })  
 .send({ age: 1 })  
 .end((err, res) => {  
 res.should.be.json();  
 res.text.should.equal('{"name":"tobi","age":1}');  
 ({ name: 'tobi', age: 1 }.should.eql(res.body));  
 done();  
 });  
 }

# res.body

# application/json

should parse the body

done => {  
 request.get(`${uri}/json`).end((err, res) => {  
 res.text.should.equal('{"name":"manny"}');  
 res.body.should.eql({ name: 'manny' });  
 done();  
 });  
 }

# HEAD requests

should not throw a parse error

done => {  
 request.head(`${uri}/json`).end((err, res) => {  
 try {  
 assert.strictEqual(err, null);  
 assert.strictEqual(res.text, undefined);  
 assert.strictEqual(Object.keys(res.body).length, 0);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# Invalid JSON response

should return the raw response

done => {  
 request.get(`${uri}/invalid-json`).end((err, res) => {  
 assert.deepEqual(  
 err.rawResponse,  
 ")]}', {'header':{'code':200,'text':'OK','version':'1.0'},'data':'some data'}"  
 );  
 done();  
 });  
 }

should return the http status code

done => {  
 request.get(`${uri}/invalid-json-forbidden`).end((err, res) => {  
 assert.equal(err.statusCode, 403);  
 done();  
 });  
 }

# No content

should not throw a parse error

done => {  
 request.get(`${uri}/no-content`).end((err, res) => {  
 try {  
 assert.strictEqual(err, null);  
 assert.strictEqual(res.text, '');  
 assert.strictEqual(Object.keys(res.body).length, 0);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# application/json+hal

should parse the body

done => {  
 request.get(`${uri}/json-hal`).end((err, res) => {  
 if (err) return done(err);  
 res.text.should.equal('{"name":"hal 5000"}');  
 res.body.should.eql({ name: 'hal 5000' });  
 done();  
 });  
 }

# vnd.collection+json

should parse the body

done => {  
 request.get(`${uri}/collection-json`).end((err, res) => {  
 res.text.should.equal('{"name":"chewbacca"}');  
 res.body.should.eql({ name: 'chewbacca' });  
 done();  
 });  
 }

# request

# on redirect

should retain header fields

done => {  
 request  
 .get(`${base}/header`)  
 .set('X-Foo', 'bar')  
 .end((err, res) => {  
 try {  
 assert(res.body);  
 res.body.should.have.property('x-foo', 'bar');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should preserve timeout across redirects

done => {  
 request  
 .get(`${base}/movies/random`)  
 .timeout(250)  
 .end((err, res) => {  
 try {  
 assert(err instanceof Error, 'expected an error');  
 err.should.have.property('timeout', 250);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should successfully redirect after retry on error

done => {  
 const id = Math.random() \* 1000000 \* Date.now();  
 request  
 .get(`${base}/error/redirect/${id}`)  
 .retry(2)  
 .end((err, res) => {  
 assert(res.ok, 'response should be ok');  
 assert(res.text, 'first movie page');  
 done();  
 });  
 }

should preserve retries across redirects

done => {  
 const id = Math.random() \* 1000000 \* Date.now();  
 request  
 .get(`${base}/error/redirect-error${id}`)  
 .retry(2)  
 .end((err, res) => {  
 assert(err, 'expected an error');  
 assert.equal(2, err.retries, 'expected an error with .retries');  
 assert.equal(500, err.status, 'expected an error status of 500');  
 done();  
 });  
 }

# on 303

should redirect with same method

done => {  
 request  
 .put(`${base}/redirect-303`)  
 .send({ msg: 'hello' })  
 .redirects(1)  
 .on('redirect', res => {  
 res.headers.location.should.equal('/reply-method');  
 })  
 .end((err, res) => {  
 if (err) {  
 done(err);  
 return;  
 }  
 res.text.should.equal('method=get');  
 done();  
 });  
 }

# on 307

should redirect with same method

done => {  
 if (isMSIE) return done(); // IE9 broken  
 request  
 .put(`${base}/redirect-307`)  
 .send({ msg: 'hello' })  
 .redirects(1)  
 .on('redirect', res => {  
 res.headers.location.should.equal('/reply-method');  
 })  
 .end((err, res) => {  
 if (err) {  
 done(err);  
 return;  
 }  
 res.text.should.equal('method=put');  
 done();  
 });  
 }

# on 308

should redirect with same method

done => {  
 if (isMSIE) return done(); // IE9 broken  
 request  
 .put(`${base}/redirect-308`)  
 .send({ msg: 'hello' })  
 .redirects(1)  
 .on('redirect', res => {  
 res.headers.location.should.equal('/reply-method');  
 })  
 .end((err, res) => {  
 if (err) {  
 done(err);  
 return;  
 }  
 res.text.should.equal('method=put');  
 done();  
 });  
 }

# request

Request inheritance

assert(request.get(`${uri}/`) instanceof request.Request);

request() simple GET without callback

next => {  
 request('GET', 'test/test.request.js').end();  
 next();  
 }

request() simple GET

next => {  
 request('GET', `${uri}/ok`).end((err, res) => {  
 try {  
 assert(res instanceof request.Response, 'respond with Response');  
 assert(res.ok, 'response should be ok');  
 assert(res.text, 'res.text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() simple HEAD

next => {  
 request.head(`${uri}/ok`).end((err, res) => {  
 try {  
 assert(res instanceof request.Response, 'respond with Response');  
 assert(res.ok, 'response should be ok');  
 assert(!res.text, 'res.text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() GET 5xx

next => {  
 request('GET', `${uri}/error`).end((err, res) => {  
 try {  
 assert(err);  
 assert.equal(err.message, 'Internal Server Error');  
 assert(!res.ok, 'response should not be ok');  
 assert(res.error, 'response should be an error');  
 assert(!res.clientError, 'response should not be a client error');  
 assert(res.serverError, 'response should be a server error');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() GET 4xx

next => {  
 request('GET', `${uri}/notfound`).end((err, res) => {  
 try {  
 assert(err);  
 assert.equal(err.message, 'Not Found');  
 assert(!res.ok, 'response should not be ok');  
 assert(res.error, 'response should be an error');  
 assert(res.clientError, 'response should be a client error');  
 assert(!res.serverError, 'response should not be a server error');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() GET 404 Not Found

next => {  
 request('GET', `${uri}/notfound`).end((err, res) => {  
 try {  
 assert(err);  
 assert(res.notFound, 'response should be .notFound');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() GET 400 Bad Request

next => {  
 request('GET', `${uri}/bad-request`).end((err, res) => {  
 try {  
 assert(err);  
 assert(res.badRequest, 'response should be .badRequest');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() GET 401 Bad Request

next => {  
 request('GET', `${uri}/unauthorized`).end((err, res) => {  
 try {  
 assert(err);  
 assert(res.unauthorized, 'response should be .unauthorized');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() GET 406 Not Acceptable

next => {  
 request('GET', `${uri}/not-acceptable`).end((err, res) => {  
 try {  
 assert(err);  
 assert(res.notAcceptable, 'response should be .notAcceptable');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() GET 204 No Content

next => {  
 request('GET', `${uri}/no-content`).end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(res.noContent, 'response should be .noContent');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() DELETE 204 No Content

next => {  
 request('DELETE', `${uri}/no-content`).end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(res.noContent, 'response should be .noContent');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() header parsing

next => {  
 request('GET', `${uri}/notfound`).end((err, res) => {  
 try {  
 assert(err);  
 assert.equal('text/html; charset=utf-8', res.header['content-type']);  
 assert.equal('Express', res.header['x-powered-by']);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request() .status

next => {  
 request('GET', `${uri}/notfound`).end((err, res) => {  
 try {  
 assert(err);  
 assert.equal(404, res.status, 'response .status');  
 assert.equal(4, res.statusType, 'response .statusType');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

get()

next => {  
 request.get(`${uri}/notfound`).end((err, res) => {  
 try {  
 assert(err);  
 assert.equal(404, res.status, 'response .status');  
 assert.equal(4, res.statusType, 'response .statusType');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

put()

next => {  
 request.put(`${uri}/user/12`).end((err, res) => {  
 try {  
 assert.equal('updated', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

put().send()

next => {  
 request  
 .put(`${uri}/user/13/body`)  
 .send({ user: 'new' })  
 .end((err, res) => {  
 try {  
 assert.equal('received new', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

post()

next => {  
 request.post(`${uri}/user`).end((err, res) => {  
 try {  
 assert.equal('created', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

del()

next => {  
 request.del(`${uri}/user/12`).end((err, res) => {  
 try {  
 assert.equal('deleted', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

delete()

next => {  
 request.delete(`${uri}/user/12`).end((err, res) => {  
 try {  
 assert.equal('deleted', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

post() data

next => {  
 request  
 .post(`${uri}/todo/item`)  
 .type('application/octet-stream')  
 .send('tobi')  
 .end((err, res) => {  
 try {  
 assert.equal('added "tobi"', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .type()

next => {  
 request  
 .post(`${uri}/user/12/pet`)  
 .type('urlencoded')  
 .send('pet=tobi')  
 .end((err, res) => {  
 try {  
 assert.equal('added pet "tobi"', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .type() with alias

next => {  
 request  
 .post(`${uri}/user/12/pet`)  
 .type('application/x-www-form-urlencoded')  
 .send('pet=tobi')  
 .end((err, res) => {  
 try {  
 assert.equal('added pet "tobi"', res.text, 'response text');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .get() with no data or callback

next => {  
 request.get(`${uri}/echo-header/content-type`);  
 next();  
 }

request .send() with no data only

next => {  
 request  
 .post(`${uri}/user/5/pet`)  
 .type('urlencoded')  
 .send('pet=tobi');  
 next();  
 }

request .send() with callback only

next => {  
 request  
 .get(`${uri}/echo-header/accept`)  
 .set('Accept', 'foo/bar')  
 .end((err, res) => {  
 try {  
 assert.equal('foo/bar', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .accept() with json

next => {  
 request  
 .get(`${uri}/echo-header/accept`)  
 .accept('json')  
 .end((err, res) => {  
 try {  
 assert.equal('application/json', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .accept() with application/json

next => {  
 request  
 .get(`${uri}/echo-header/accept`)  
 .accept('application/json')  
 .end((err, res) => {  
 try {  
 assert.equal('application/json', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .accept() with xml

next => {  
 request  
 .get(`${uri}/echo-header/accept`)  
 .accept('xml')  
 .end((err, res) => {  
 try {  
 // We can't depend on mime module to be consistent with this  
 assert(res.text == 'application/xml' || res.text == 'text/xml');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .accept() with application/xml

next => {  
 request  
 .get(`${uri}/echo-header/accept`)  
 .accept('application/xml')  
 .end((err, res) => {  
 try {  
 assert.equal('application/xml', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .end()

next => {  
 request  
 .put(`${uri}/echo-header/content-type`)  
 .set('Content-Type', 'text/plain')  
 .send('wahoo')  
 .end((err, res) => {  
 try {  
 assert.equal('text/plain', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .send()

next => {  
 request  
 .put(`${uri}/echo-header/content-type`)  
 .set('Content-Type', 'text/plain')  
 .send('wahoo')  
 .end((err, res) => {  
 try {  
 assert.equal('text/plain', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .set()

next => {  
 request  
 .put(`${uri}/echo-header/content-type`)  
 .set('Content-Type', 'text/plain')  
 .send('wahoo')  
 .end((err, res) => {  
 try {  
 assert.equal('text/plain', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request .set(object)

next => {  
 request  
 .put(`${uri}/echo-header/content-type`)  
 .set({ 'Content-Type': 'text/plain' })  
 .send('wahoo')  
 .end((err, res) => {  
 try {  
 assert.equal('text/plain', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST urlencoded

next => {  
 request  
 .post(`${uri}/pet`)  
 .type('urlencoded')  
 .send({ name: 'Manny', species: 'cat' })  
 .end((err, res) => {  
 try {  
 assert.equal('added Manny the cat', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST json

next => {  
 request  
 .post(`${uri}/pet`)  
 .type('json')  
 .send({ name: 'Manny', species: 'cat' })  
 .end((err, res) => {  
 try {  
 assert.equal('added Manny the cat', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST json array

next => {  
 request  
 .post(`${uri}/echo`)  
 .send([1, 2, 3])  
 .end((err, res) => {  
 try {  
 assert.equal(  
 'application/json',  
 res.header['content-type'].split(';')[0]  
 );  
 assert.equal('[1,2,3]', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST json default

next => {  
 request  
 .post(`${uri}/pet`)  
 .send({ name: 'Manny', species: 'cat' })  
 .end((err, res) => {  
 try {  
 assert.equal('added Manny the cat', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST json contentType charset

next => {  
 request  
 .post(`${uri}/echo`)  
 .set('Content-Type', 'application/json; charset=UTF-8')  
 .send({ data: ['data1', 'data2'] })  
 .end((err, res) => {  
 try {  
 assert.equal('{"data":["data1","data2"]}', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST json contentType vendor

next => {  
 request  
 .post(`${uri}/echo`)  
 .set('Content-Type', 'application/vnd.example+json')  
 .send({ data: ['data1', 'data2'] })  
 .end((err, res) => {  
 try {  
 assert.equal('{"data":["data1","data2"]}', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST multiple .send() calls

next => {  
 request  
 .post(`${uri}/pet`)  
 .send({ name: 'Manny' })  
 .send({ species: 'cat' })  
 .end((err, res) => {  
 try {  
 assert.equal('added Manny the cat', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST multiple .send() strings

next => {  
 request  
 .post(`${uri}/echo`)  
 .send('user[name]=tj')  
 .send('user[email]=tj@vision-media.ca')  
 .end((err, res) => {  
 try {  
 assert.equal(  
 'application/x-www-form-urlencoded',  
 res.header['content-type'].split(';')[0]  
 );  
 assert.equal(  
 res.text,  
 'user[name]=tj&user[email]=tj@vision-media.ca'  
 );  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST with no data

next => {  
 request  
 .post(`${uri}/empty-body`)  
 .send()  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(res.noContent, 'response should be .noContent');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET .type

next => {  
 request.get(`${uri}/pets`).end((err, res) => {  
 try {  
 assert.equal('application/json', res.type);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET Content-Type params

next => {  
 request.get(`${uri}/text`).end((err, res) => {  
 try {  
 assert.equal('utf-8', res.charset);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET json

next => {  
 request.get(`${uri}/pets`).end((err, res) => {  
 try {  
 assert.deepEqual(res.body, ['tobi', 'loki', 'jane']);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET json-seq

next => {  
 request  
 .get(`${uri}/json-seq`)  
 .buffer()  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert.deepEqual(res.text, '\u001E{"id":1}\n\u001E{"id":2}\n');  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET x-www-form-urlencoded

next => {  
 request.get(`${uri}/foo`).end((err, res) => {  
 try {  
 assert.deepEqual(res.body, { foo: 'bar' });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET shorthand

next => {  
 request.get(`${uri}/foo`, (err, res) => {  
 try {  
 assert.equal('foo=bar', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST shorthand

next => {  
 request.post(`${uri}/user/0/pet`, { pet: 'tobi' }, (err, res) => {  
 try {  
 assert.equal('added pet "tobi"', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

POST shorthand without callback

next => {  
 request.post(`${uri}/user/0/pet`, { pet: 'tobi' }).end((err, res) => {  
 try {  
 assert.equal('added pet "tobi"', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring object with array

next => {  
 request  
 .get(`${uri}/querystring`)  
 .query({ val: ['a', 'b', 'c'] })  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, { val: ['a', 'b', 'c'] });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring object with array and primitives

next => {  
 request  
 .get(`${uri}/querystring`)  
 .query({ array: ['a', 'b', 'c'], string: 'foo', number: 10 })  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, {  
 array: ['a', 'b', 'c'],  
 string: 'foo',  
 number: 10  
 });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring object with two arrays

next => {  
 request  
 .get(`${uri}/querystring`)  
 .query({ array1: ['a', 'b', 'c'], array2: [1, 2, 3] })  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, {  
 array1: ['a', 'b', 'c'],  
 array2: [1, 2, 3]  
 });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring object

next => {  
 request  
 .get(`${uri}/querystring`)  
 .query({ search: 'Manny' })  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, { search: 'Manny' });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring append original

next => {  
 request  
 .get(`${uri}/querystring?search=Manny`)  
 .query({ range: '1..5' })  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, { search: 'Manny', range: '1..5' });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring multiple objects

next => {  
 request  
 .get(`${uri}/querystring`)  
 .query({ search: 'Manny' })  
 .query({ range: '1..5' })  
 .query({ order: 'desc' })  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, {  
 search: 'Manny',  
 range: '1..5',  
 order: 'desc'  
 });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring with strings

next => {  
 request  
 .get(`${uri}/querystring`)  
 .query('search=Manny')  
 .query('range=1..5')  
 .query('order=desc')  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, {  
 search: 'Manny',  
 range: '1..5',  
 order: 'desc'  
 });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET querystring with strings and objects

next => {  
 request  
 .get(`${uri}/querystring`)  
 .query('search=Manny')  
 .query({ order: 'desc', range: '1..5' })  
 .end((err, res) => {  
 try {  
 assert.deepEqual(res.body, {  
 search: 'Manny',  
 range: '1..5',  
 order: 'desc'  
 });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

GET shorthand payload goes to querystring

next => {  
 request.get(  
 `${uri}/querystring`,  
 { foo: 'FOO', bar: 'BAR' },  
 (err, res) => {  
 try {  
 assert.deepEqual(res.body, { foo: 'FOO', bar: 'BAR' });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 }  
 );  
 }

HEAD shorthand payload goes to querystring

next => {  
 request.head(  
 `${uri}/querystring-in-header`,  
 { foo: 'FOO', bar: 'BAR' },  
 (err, res) => {  
 try {  
 assert.deepEqual(JSON.parse(res.headers.query), {  
 foo: 'FOO',  
 bar: 'BAR'  
 });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 }  
 );  
 }

request(method, url)

next => {  
 request('GET', `${uri}/foo`).end((err, res) => {  
 try {  
 assert.equal('bar', res.body.foo);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request(url)

next => {  
 request(`${uri}/foo`).end((err, res) => {  
 try {  
 assert.equal('bar', res.body.foo);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request(url, fn)

next => {  
 request(`${uri}/foo`, (err, res) => {  
 try {  
 assert.equal('bar', res.body.foo);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

req.timeout(ms)

next => {  
 const req = request.get(`${uri}/delay/3000`).timeout(1000);  
 req.end((err, res) => {  
 try {  
 assert(err, 'error missing');  
 assert.equal(1000, err.timeout, 'err.timeout missing');  
 assert.equal(  
 'Timeout of 1000ms exceeded',  
 err.message,  
 'err.message incorrect'  
 );  
 assert.equal(null, res);  
 assert(req.timedout, true);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

req.timeout(ms) with redirect

next => {  
 const req = request.get(`${uri}/delay/const`).timeout(1000);  
 req.end((err, res) => {  
 try {  
 assert(err, 'error missing');  
 assert.equal(1000, err.timeout, 'err.timeout missing');  
 assert.equal(  
 'Timeout of 1000ms exceeded',  
 err.message,  
 'err.message incorrect'  
 );  
 assert.equal(null, res);  
 assert(req.timedout, true);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

request event

next => {  
 request  
 .get(`${uri}/foo`)  
 .on('request', req => {  
 try {  
 assert.equal(`${uri}/foo`, req.url);  
 next();  
 } catch (err) {  
 next(err);  
 }  
 })  
 .end();  
 }

response event

next => {  
 request  
 .get(`${uri}/foo`)  
 .on('response', res => {  
 try {  
 assert.equal('bar', res.body.foo);  
 next();  
 } catch (err) {  
 next(err);  
 }  
 })  
 .end();  
 }

response should set statusCode

next => {  
 request.get(`${uri}/ok`, (err, res) => {  
 try {  
 assert.strictEqual(res.statusCode, 200);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

req.toJSON()

next => {  
 request.get(`${uri}/ok`).end((err, res) => {  
 try {  
 const j = (res.request || res.req).toJSON();  
 ['url', 'method', 'data', 'headers'].forEach(prop => {  
 assert(j.hasOwnProperty(prop));  
 });  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

# .retry(count)

should not retry if passed "0"

done => {  
 request  
 .get(`${base}/error`)  
 .retry(0)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(  
 undefined,  
 err.retries,  
 'expected an error without .retries'  
 );  
 assert.equal(500, err.status, 'expected an error status of 500');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should not retry if passed an invalid number

done => {  
 request  
 .get(`${base}/error`)  
 .retry(-2)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(  
 undefined,  
 err.retries,  
 'expected an error without .retries'  
 );  
 assert.equal(500, err.status, 'expected an error status of 500');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should not retry if passed undefined

done => {  
 request  
 .get(`${base}/error`)  
 .retry(undefined)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(  
 undefined,  
 err.retries,  
 'expected an error without .retries'  
 );  
 assert.equal(500, err.status, 'expected an error status of 500');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should handle server error after repeat attempt

done => {  
 request  
 .get(`${base}/error`)  
 .retry(2)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(2, err.retries, 'expected an error with .retries');  
 assert.equal(500, err.status, 'expected an error status of 500');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should retry if passed nothing

done => {  
 request  
 .get(`${base}/error`)  
 .retry()  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(1, err.retries, 'expected an error with .retries');  
 assert.equal(500, err.status, 'expected an error status of 500');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should retry if passed "true"

done => {  
 request  
 .get(`${base}/error`)  
 .retry(true)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(1, err.retries, 'expected an error with .retries');  
 assert.equal(500, err.status, 'expected an error status of 500');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should handle successful request after repeat attempt from server error

done => {  
 request  
 .get(`${base}/error/ok/${uniqid()}`)  
 .query({ qs: 'present' })  
 .retry(2)  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(res.ok, 'response should be ok');  
 assert(res.text, 'res.text');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should handle server timeout error after repeat attempt

done => {  
 request  
 .get(`${base}/delay/400`)  
 .timeout(200)  
 .retry(2)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(2, err.retries, 'expected an error with .retries');  
 assert.equal(  
 'number',  
 typeof err.timeout,  
 'expected an error with .timeout'  
 );  
 assert.equal('ECONNABORTED', err.code, 'expected abort error code');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should handle successful request after repeat attempt from server timeout

done => {  
 const url = `/delay/1200/ok/${uniqid()}?built=in`;  
 request  
 .get(base + url)  
 .query('string=ified')  
 .query({ json: 'ed' })  
 .timeout(600)  
 .retry(2)  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(res.ok, 'response should be ok');  
 assert.equal(res.text, `ok = ${url}&string=ified&json=ed`);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should correctly abort a retry attempt

done => {  
 let aborted = false;  
 const req = request  
 .get(`${base}/delay/400`)  
 .timeout(200)  
 .retry(2);  
 req.end((err, res) => {  
 try {  
 assert(false, 'should not complete the request');  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 req.on('abort', () => {  
 aborted = true;  
 });  
 setTimeout(() => {  
 req.abort();  
 setTimeout(() => {  
 try {  
 assert(aborted, 'should be aborted');  
 done();  
 } catch (err) {  
 done(err);  
 }  
 }, 150);  
 }, 150);  
 }

should correctly retain header fields

done => {  
 request  
 .get(`${base}/error/ok/${uniqid()}`)  
 .query({ qs: 'present' })  
 .retry(2)  
 .set('X-Foo', 'bar')  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(res.body);  
 res.body.should.have.property('x-foo', 'bar');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should not retry on 4xx responses

done => {  
 request  
 .get(`${base}/bad-request`)  
 .retry(2)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(0, err.retries, 'expected an error with 0 .retries');  
 assert.equal(400, err.status, 'expected an error status of 400');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should execute callback on retry if passed

done => {  
 let callbackCallCount = 0;  
 function retryCallback(request) {  
 callbackCallCount++;  
 }  
 request  
 .get(`${base}/error`)  
 .retry(2, retryCallback)  
 .end((err, res) => {  
 try {  
 assert(err, 'expected an error');  
 assert.equal(2, err.retries, 'expected an error with .retries');  
 assert.equal(500, err.status, 'expected an error status of 500');  
 assert.equal(  
 2,  
 callbackCallCount,  
 'expected the callback to be called on each retry'  
 );  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# .timeout(ms)

# when timeout is exceeded

should error

done => {  
 request  
 .get(`${base}/delay/500`)  
 .timeout(150)  
 .end((err, res) => {  
 assert(err, 'expected an error');  
 assert.equal(  
 'number',  
 typeof err.timeout,  
 'expected an error with .timeout'  
 );  
 assert.equal('ECONNABORTED', err.code, 'expected abort error code');  
 done();  
 });  
 }

should handle gzip timeout

done => {  
 request  
 .get(`${base}/delay/zip`)  
 .timeout(150)  
 .end((err, res) => {  
 assert(err, 'expected an error');  
 assert.equal(  
 'number',  
 typeof err.timeout,  
 'expected an error with .timeout'  
 );  
 assert.equal('ECONNABORTED', err.code, 'expected abort error code');  
 done();  
 });  
 }

should handle buffer timeout

done => {  
 request  
 .get(`${base}/delay/json`)  
 .buffer(true)  
 .timeout(150)  
 .end((err, res) => {  
 assert(err, 'expected an error');  
 assert.equal(  
 'number',  
 typeof err.timeout,  
 'expected an error with .timeout'  
 );  
 assert.equal('ECONNABORTED', err.code, 'expected abort error code');  
 done();  
 });  
 }

should error on deadline

done => {  
 request  
 .get(`${base}/delay/500`)  
 .timeout({ deadline: 150 })  
 .end((err, res) => {  
 assert(err, 'expected an error');  
 assert.equal(  
 'number',  
 typeof err.timeout,  
 'expected an error with .timeout'  
 );  
 assert.equal('ECONNABORTED', err.code, 'expected abort error code');  
 done();  
 });  
 }

should support setting individual options

done => {  
 request  
 .get(`${base}/delay/500`)  
 .timeout({ deadline: 10 })  
 .timeout({ response: 99999 })  
 .end((err, res) => {  
 assert(err, 'expected an error');  
 assert.equal('ECONNABORTED', err.code, 'expected abort error code');  
 assert.equal('ETIME', err.errno);  
 done();  
 });  
 }

should error on response

done => {  
 request  
 .get(`${base}/delay/500`)  
 .timeout({ response: 150 })  
 .end((err, res) => {  
 assert(err, 'expected an error');  
 assert.equal(  
 'number',  
 typeof err.timeout,  
 'expected an error with .timeout'  
 );  
 assert.equal('ECONNABORTED', err.code, 'expected abort error code');  
 assert.equal('ETIMEDOUT', err.errno);  
 done();  
 });  
 }

should accept slow body with fast response

done => {  
 request  
 .get(`${base}/delay/slowbody`)  
 .timeout({ response: 1000 })  
 .on('progress', () => {  
 // This only makes the test faster without relying on arbitrary timeouts  
 request.get(`${base}/delay/slowbody/finish`).end();  
 })  
 .end(done);  
 }

# request

# use

should use plugin success

done => {  
 const now = `${Date.now()}`;  
 function uuid(req) {  
 req.set('X-UUID', now);  
 return req;  
 }  
 function prefix(req) {  
 req.url = uri + req.url;  
 return req;  
 }  
 request  
 .get('/echo')  
 .use(uuid)  
 .use(prefix)  
 .end((err, res) => {  
 assert.strictEqual(res.statusCode, 200);  
 assert.equal(res.get('X-UUID'), now);  
 done();  
 });  
 }

# subclass

should be an instance of Request

const req = request.get('/');  
assert(req instanceof request.Request);

should use patched subclass

assert(OriginalRequest);  
let constructorCalled;  
let sendCalled;  
function NewRequest(...args) {  
 constructorCalled = true;  
 OriginalRequest.apply(this, args);  
}  
NewRequest.prototype = Object.create(OriginalRequest.prototype);  
NewRequest.prototype.send = function() {  
 sendCalled = true;  
 return this;  
};  
request.Request = NewRequest;  
const req = request.get('/').send();  
assert(constructorCalled);  
assert(sendCalled);  
assert(req instanceof NewRequest);  
assert(req instanceof OriginalRequest);

should use patched subclass in agent too

if (!request.agent) return; // Node-only  
function NewRequest(...args) {  
 OriginalRequest.apply(this, args);  
}  
NewRequest.prototype = Object.create(OriginalRequest.prototype);  
request.Request = NewRequest;  
const req = request.agent().del('/');  
assert(req instanceof NewRequest);  
assert(req instanceof OriginalRequest);

# request

# persistent agent

should gain a session on POST

agent3.post(`${base}/signin`).then(res => {  
 res.should.have.status(200);  
 should.not.exist(res.headers['set-cookie']);  
 res.text.should.containEql('dashboard');  
 })

should start with empty session (set cookies)

done => {  
 agent1.get(`${base}/dashboard`).end((err, res) => {  
 should.exist(err);  
 res.should.have.status(401);  
 should.exist(res.headers['set-cookie']);  
 done();  
 });  
 }

should gain a session (cookies already set)

agent1.post(`${base}/signin`).then(res => {  
 res.should.have.status(200);  
 should.not.exist(res.headers['set-cookie']);  
 res.text.should.containEql('dashboard');  
 })

should persist cookies across requests

agent1.get(`${base}/dashboard`).then(res => {  
 res.should.have.status(200);  
 })

should have the cookie set in the end callback

agent4  
 .post(`${base}/setcookie`)  
 .then(() => agent4.get(`${base}/getcookie`))  
 .then(res => {  
 res.should.have.status(200);  
 assert.strictEqual(res.text, 'jar');  
 })

should not share cookies

done => {  
 agent2.get(`${base}/dashboard`).end((err, res) => {  
 should.exist(err);  
 res.should.have.status(401);  
 done();  
 });  
 }

should not lose cookies between agents

agent1.get(`${base}/dashboard`).then(res => {  
 res.should.have.status(200);  
 })

should be able to follow redirects

agent1.get(base).then(res => {  
 res.should.have.status(200);  
 res.text.should.containEql('dashboard');  
 })

should be able to post redirects

agent1  
 .post(`${base}/redirect`)  
 .send({ foo: 'bar', baz: 'blaaah' })  
 .then(res => {  
 res.should.have.status(200);  
 res.text.should.containEql('simple');  
 res.redirects.should.eql([`${base}/simple`]);  
 })

should be able to limit redirects

done => {  
 agent1  
 .get(base)  
 .redirects(0)  
 .end((err, res) => {  
 should.exist(err);  
 res.should.have.status(302);  
 res.redirects.should.eql([]);  
 res.header.location.should.equal('/dashboard');  
 done();  
 });  
 }

should be able to create a new session (clear cookie)

agent1.post(`${base}/signout`).then(res => {  
 res.should.have.status(200);  
 should.exist(res.headers['set-cookie']);  
 })

should regenerate with an empty session

done => {  
 agent1.get(`${base}/dashboard`).end((err, res) => {  
 should.exist(err);  
 res.should.have.status(401);  
 should.not.exist(res.headers['set-cookie']);  
 done();  
 });  
 }

# Basic auth

# when credentials are present in url

should set Authorization

done => {  
 const new\_url = URL.parse(base);  
 new\_url.auth = 'tobi:learnboost';  
 new\_url.pathname = '/basic-auth';  
 request.get(URL.format(new\_url)).end((err, res) => {  
 res.status.should.equal(200);  
 done();  
 });  
 }

# req.auth(user, pass)

should set Authorization

done => {  
 request  
 .get(`${base}/basic-auth`)  
 .auth('tobi', 'learnboost')  
 .end((err, res) => {  
 res.status.should.equal(200);  
 done();  
 });  
 }

# req.auth(user + ":" + pass)

should set authorization

done => {  
 request  
 .get(`${base}/basic-auth/again`)  
 .auth('tobi')  
 .end((err, res) => {  
 res.status.should.eql(200);  
 done();  
 });  
 }

# [node] request

should send body with .get().send()

next => {  
 request  
 .get(`${base}/echo`)  
 .set('Content-Type', 'text/plain')  
 .send('wahoo')  
 .end((err, res) => {  
 try {  
 assert.equal('wahoo', res.text);  
 next();  
 } catch (err2) {  
 next(err2);  
 }  
 });  
 }

# with an url

should preserve the encoding of the url

done => {  
 request.get(`${base}/url?a=(b%29`).end((err, res) => {  
 assert.equal('/url?a=(b%29', res.text);  
 done();  
 });  
 }

# with an object

should format the url

request.get(url.parse(`${base}/login`)).then(res => {  
 assert(res.ok);  
 })

# without a schema

should default to http

request.get('localhost:5000/login').then(res => {  
 assert.equal(res.status, 200);  
 })

# res.toJSON()

should describe the response

request  
 .post(`${base}/echo`)  
 .send({ foo: 'baz' })  
 .then(res => {  
 const obj = res.toJSON();  
 assert.equal('object', typeof obj.header);  
 assert.equal('object', typeof obj.req);  
 assert.equal(200, obj.status);  
 assert.equal('{"foo":"baz"}', obj.text);  
 })

# res.links

should default to an empty object

request.get(`${base}/login`).then(res => {  
 res.links.should.eql({});  
 })

should parse the Link header field

done => {  
 request.get(`${base}/links`).end((err, res) => {  
 res.links.next.should.equal(  
 'https://api.github.com/repos/visionmedia/mocha/issues?page=2'  
 );  
 done();  
 });  
 }

# req.unset(field)

should remove the header field

done => {  
 request  
 .post(`${base}/echo`)  
 .unset('User-Agent')  
 .end((err, res) => {  
 assert.equal(void 0, res.header['user-agent']);  
 done();  
 });  
 }

# case-insensitive

should set/get header fields case-insensitively

const r = request.post(`${base}/echo`);  
r.set('MiXeD', 'helloes');  
assert.strictEqual(r.get('mixed'), 'helloes');

should unset header fields case-insensitively

const r = request.post(`${base}/echo`);  
r.set('MiXeD', 'helloes');  
r.unset('MIXED');  
assert.strictEqual(r.get('mixed'), undefined);

# req.write(str)

should write the given data

done => {  
 const req = request.post(`${base}/echo`);  
 req.set('Content-Type', 'application/json');  
 assert.equal('boolean', typeof req.write('{"name"'));  
 assert.equal('boolean', typeof req.write(':"tobi"}'));  
 req.end((err, res) => {  
 res.text.should.equal('{"name":"tobi"}');  
 done();  
 });  
 }

# req.pipe(stream)

should pipe the response to the given stream

done => {  
 const stream = new EventEmitter();  
 stream.buf = '';  
 stream.writable = true;  
 stream.write = function(chunk) {  
 this.buf += chunk;  
 };  
 stream.end = function() {  
 this.buf.should.equal('{"name":"tobi"}');  
 done();  
 };  
 request  
 .post(`${base}/echo`)  
 .send('{"name":"tobi"}')  
 .pipe(stream);  
 }

# .buffer()

should enable buffering

done => {  
 request  
 .get(`${base}/custom`)  
 .buffer()  
 .end((err, res) => {  
 assert.ifError(err);  
 assert.equal('custom stuff', res.text);  
 assert(res.buffered);  
 done();  
 });  
 }

should take precedence over request.buffer['someMimeType'] = false

done => {  
 const type = 'application/barbaz';  
 const send = 'some text';  
 request.buffer[type] = false;  
 request  
 .post(`${base}/echo`)  
 .type(type)  
 .send(send)  
 .buffer()  
 .end((err, res) => {  
 delete request.buffer[type];  
 assert.ifError(err);  
 assert.equal(res.type, type);  
 assert.equal(send, res.text);  
 assert(res.buffered);  
 done();  
 });  
 }

# .buffer(false)

should disable buffering

done => {  
 request  
 .post(`${base}/echo`)  
 .type('application/x-dog')  
 .send('hello this is dog')  
 .buffer(false)  
 .end((err, res) => {  
 assert.ifError(err);  
 assert.equal(null, res.text);  
 res.body.should.eql({});  
 let buf = '';  
 res.setEncoding('utf8');  
 res.on('data', chunk => {  
 buf += chunk;  
 });  
 res.on('end', () => {  
 buf.should.equal('hello this is dog');  
 done();  
 });  
 });  
 }

should take precedence over request.buffer['someMimeType'] = true

done => {  
 const type = 'application/foobar';  
 const send = 'hello this is a dog';  
 request.buffer[type] = true;  
 request  
 .post(`${base}/echo`)  
 .type(type)  
 .send(send)  
 .buffer(false)  
 .end((err, res) => {  
 delete request.buffer[type];  
 assert.ifError(err);  
 assert.equal(null, res.text);  
 assert.equal(res.type, type);  
 assert(!res.buffered);  
 res.body.should.eql({});  
 let buf = '';  
 res.setEncoding('utf8');  
 res.on('data', chunk => {  
 buf += chunk;  
 });  
 res.on('end', () => {  
 buf.should.equal(send);  
 done();  
 });  
 });  
 }

# .withCredentials()

should not throw an error when using the client-side "withCredentials" method

done => {  
 request  
 .get(`${base}/custom`)  
 .withCredentials()  
 .end((err, res) => {  
 assert.ifError(err);  
 done();  
 });  
 }

# .agent()

should return the defaut agent

done => {  
 const req = request.post(`${base}/echo`);  
 req.agent().should.equal(false);  
 done();  
 }

# .agent(undefined)

should set an agent to undefined and ensure it is chainable

done => {  
 const req = request.get(`${base}/echo`);  
 const ret = req.agent(undefined);  
 ret.should.equal(req);  
 assert.strictEqual(req.agent(), undefined);  
 done();  
 }

# .agent(new http.Agent())

should set passed agent

done => {  
 const http = require('http');  
 const req = request.get(`${base}/echo`);  
 const agent = new http.Agent();  
 const ret = req.agent(agent);  
 ret.should.equal(req);  
 req.agent().should.equal(agent);  
 done();  
 }

# with a content type other than application/json or text/\*

should still use buffering

return request  
 .post(`${base}/echo`)  
 .type('application/x-dog')  
 .send('hello this is dog')  
 .then(res => {  
 assert.equal(null, res.text);  
 assert.equal(res.body.toString(), 'hello this is dog');  
 res.buffered.should.be.true;  
 });

# content-length

should be set to the byte length of a non-buffer object

done => {  
 const decoder = new StringDecoder('utf8');  
 let img = fs.readFileSync(`${\_\_dirname}/fixtures/test.png`);  
 img = decoder.write(img);  
 request  
 .post(`${base}/echo`)  
 .type('application/x-image')  
 .send(img)  
 .buffer(false)  
 .end((err, res) => {  
 assert.ifError(err);  
 assert(!res.buffered);  
 assert.equal(res.header['content-length'], Buffer.byteLength(img));  
 done();  
 });  
 }

should be set to the length of a buffer object

done => {  
 const img = fs.readFileSync(`${\_\_dirname}/fixtures/test.png`);  
 request  
 .post(`${base}/echo`)  
 .type('application/x-image')  
 .send(img)  
 .buffer(true)  
 .end((err, res) => {  
 assert.ifError(err);  
 assert(res.buffered);  
 assert.equal(res.header['content-length'], img.length);  
 done();  
 });  
 }

# req.buffer['someMimeType']

should respect that agent.buffer(true) takes precedent

done => {  
 const agent = request.agent();  
 agent.buffer(true);  
 const type = 'application/somerandomtype';  
 const send = 'somerandomtext';  
 request.buffer[type] = false;  
 agent  
 .post(`${base}/echo`)  
 .type(type)  
 .send(send)  
 .end((err, res) => {  
 delete request.buffer[type];  
 assert.ifError(err);  
 assert.equal(res.type, type);  
 assert.equal(send, res.text);  
 assert(res.buffered);  
 done();  
 });  
 }

should respect that agent.buffer(false) takes precedent

done => {  
 const agent = request.agent();  
 agent.buffer(false);  
 const type = 'application/barrr';  
 const send = 'some random text2';  
 request.buffer[type] = true;  
 agent  
 .post(`${base}/echo`)  
 .type(type)  
 .send(send)  
 .end((err, res) => {  
 delete request.buffer[type];  
 assert.ifError(err);  
 assert.equal(null, res.text);  
 assert.equal(res.type, type);  
 assert(!res.buffered);  
 res.body.should.eql({});  
 let buf = '';  
 res.setEncoding('utf8');  
 res.on('data', chunk => {  
 buf += chunk;  
 });  
 res.on('end', () => {  
 buf.should.equal(send);  
 done();  
 });  
 });  
 }

should disable buffering for that mimetype when false

done => {  
 const type = 'application/bar';  
 const send = 'some random text';  
 request.buffer[type] = false;  
 request  
 .post(`${base}/echo`)  
 .type(type)  
 .send(send)  
 .end((err, res) => {  
 delete request.buffer[type];  
 assert.ifError(err);  
 assert.equal(null, res.text);  
 assert.equal(res.type, type);  
 assert(!res.buffered);  
 res.body.should.eql({});  
 let buf = '';  
 res.setEncoding('utf8');  
 res.on('data', chunk => {  
 buf += chunk;  
 });  
 res.on('end', () => {  
 buf.should.equal(send);  
 done();  
 });  
 });  
 }

should enable buffering for that mimetype when true

done => {  
 const type = 'application/baz';  
 const send = 'woooo';  
 request.buffer[type] = true;  
 request  
 .post(`${base}/echo`)  
 .type(type)  
 .send(send)  
 .end((err, res) => {  
 delete request.buffer[type];  
 assert.ifError(err);  
 assert.equal(res.type, type);  
 assert.equal(send, res.text);  
 assert(res.buffered);  
 done();  
 });  
 }

should fallback to default handling for that mimetype when undefined

const type = 'application/bazzz';  
const send = 'woooooo';  
return request  
 .post(`${base}/echo`)  
 .type(type)  
 .send(send)  
 .then(res => {  
 assert.equal(res.type, type);  
 assert.equal(send, res.body.toString());  
 assert(res.buffered);  
 });

# exports

should expose .protocols

Object.keys(request.protocols).should.eql(['http:', 'https:', 'http2:']);

should expose .serialize

Object.keys(request.serialize).should.eql([  
 'application/x-www-form-urlencoded',  
 'application/json'  
]);

should expose .parse

Object.keys(request.parse).should.eql([  
 'application/x-www-form-urlencoded',  
 'application/json',  
 'text',  
 'application/octet-stream',  
 'application/pdf',  
 'image'  
]);

should export .buffer

Object.keys(request.buffer).should.eql([]);

# flags

# with 4xx response

should set res.error and res.clientError

done => {  
 request.get(`${base}/notfound`).end((err, res) => {  
 assert(err);  
 assert(!res.ok, 'response should not be ok');  
 assert(res.error, 'response should be an error');  
 assert(res.clientError, 'response should be a client error');  
 assert(!res.serverError, 'response should not be a server error');  
 done();  
 });  
 }

# with 5xx response

should set res.error and res.serverError

done => {  
 request.get(`${base}/error`).end((err, res) => {  
 assert(err);  
 assert(!res.ok, 'response should not be ok');  
 assert(!res.notFound, 'response should not be notFound');  
 assert(res.error, 'response should be an error');  
 assert(!res.clientError, 'response should not be a client error');  
 assert(res.serverError, 'response should be a server error');  
 done();  
 });  
 }

# with 404 Not Found

should res.notFound

done => {  
 request.get(`${base}/notfound`).end((err, res) => {  
 assert(err);  
 assert(res.notFound, 'response should be .notFound');  
 done();  
 });  
 }

# with 400 Bad Request

should set req.badRequest

done => {  
 request.get(`${base}/bad-request`).end((err, res) => {  
 assert(err);  
 assert(res.badRequest, 'response should be .badRequest');  
 done();  
 });  
 }

# with 401 Bad Request

should set res.unauthorized

done => {  
 request.get(`${base}/unauthorized`).end((err, res) => {  
 assert(err);  
 assert(res.unauthorized, 'response should be .unauthorized');  
 done();  
 });  
 }

# with 406 Not Acceptable

should set res.notAcceptable

done => {  
 request.get(`${base}/not-acceptable`).end((err, res) => {  
 assert(err);  
 assert(res.notAcceptable, 'response should be .notAcceptable');  
 done();  
 });  
 }

# with 204 No Content

should set res.noContent

done => {  
 request.get(`${base}/no-content`).end((err, res) => {  
 assert(!err);  
 assert(res.noContent, 'response should be .noContent');  
 done();  
 });  
 }

# with 201 Created

should set res.created

done => {  
 request.post(`${base}/created`).end((err, res) => {  
 assert(!err);  
 assert(res.created, 'response should be .created');  
 done();  
 });  
 }

# with 422 Unprocessable Entity

should set res.unprocessableEntity

done => {  
 request.post(`${base}/unprocessable-entity`).end((err, res) => {  
 assert(err);  
 assert(  
 res.unprocessableEntity,  
 'response should be .unprocessableEntity'  
 );  
 done();  
 });  
 }

# Merging objects

Don't mix Buffer and JSON

assert.throws(() => {  
 request  
 .post('/echo')  
 .send(Buffer.from('some buffer'))  
 .send({ allowed: false });  
});

# req.send(String)

should default to "form"

done => {  
 request  
 .post(`${base}/echo`)  
 .send('user[name]=tj')  
 .send('user[email]=tj@vision-media.ca')  
 .end((err, res) => {  
 res.header['content-type'].should.equal(  
 'application/x-www-form-urlencoded'  
 );  
 res.body.should.eql({  
 user: { name: 'tj', email: 'tj@vision-media.ca' }  
 });  
 done();  
 });  
 }

# res.body

# application/x-www-form-urlencoded

should parse the body

done => {  
 request.get(`${base}/form-data`).end((err, res) => {  
 res.text.should.equal('pet[name]=manny');  
 res.body.should.eql({ pet: { name: 'manny' } });  
 done();  
 });  
 }

# https

# certificate authority

# request

should give a good response

done => {  
 request  
 .get(testEndpoint)  
 .ca(ca)  
 .end((err, res) => {  
 assert.ifError(err);  
 assert(res.ok);  
 assert.strictEqual('Safe and secure!', res.text);  
 done();  
 });  
 }

should reject unauthorized response

return request  
 .get(testEndpoint)  
 .trustLocalhost(false)  
 .then(  
 () => {  
 throw new Error('Allows MITM');  
 },  
 () => {}  
 );

should trust localhost unauthorized response

return request.get(testEndpoint).trustLocalhost(true);

should trust overriden localhost unauthorized response

return request  
 .get(`https://example.com:${server.address().port}`)  
 .connect('127.0.0.1')  
 .trustLocalhost();

# .agent

should be able to make multiple requests without redefining the certificate

done => {  
 const agent = request.agent({ ca });  
 agent.get(testEndpoint).end((err, res) => {  
 assert.ifError(err);  
 assert(res.ok);  
 assert.strictEqual('Safe and secure!', res.text);  
 agent.get(url.parse(testEndpoint)).end((err, res) => {  
 assert.ifError(err);  
 assert(res.ok);  
 assert.strictEqual('Safe and secure!', res.text);  
 done();  
 });  
 });  
 }

# client certificates

# request

# .agent

# res.body

# image/png

should parse the body

done => {  
 request.get(`${base}/image`).end((err, res) => {  
 res.type.should.equal('image/png');  
 Buffer.isBuffer(res.body).should.be.true();  
 (res.body.length - img.length).should.equal(0);  
 done();  
 });  
 }

# application/octet-stream

should parse the body

done => {  
 request  
 .get(`${base}/image-as-octets`)  
 .buffer(true) // that's tech debt :(  
 .end((err, res) => {  
 res.type.should.equal('application/octet-stream');  
 Buffer.isBuffer(res.body).should.be.true();  
 (res.body.length - img.length).should.equal(0);  
 done();  
 });  
 }

# application/octet-stream

should parse the body (using responseType)

done => {  
 request  
 .get(`${base}/image-as-octets`)  
 .responseType('blob')  
 .end((err, res) => {  
 res.type.should.equal('application/octet-stream');  
 Buffer.isBuffer(res.body).should.be.true();  
 (res.body.length - img.length).should.equal(0);  
 done();  
 });  
 }

# zlib

should deflate the content

done => {  
 request.get(base).end((err, res) => {  
 res.should.have.status(200);  
 res.text.should.equal(subject);  
 res.headers['content-length'].should.be.below(subject.length);  
 done();  
 });  
 }

should protect from zip bombs

done => {  
 request  
 .get(base)  
 .buffer(true)  
 .maxResponseSize(1)  
 .end((err, res) => {  
 try {  
 assert.equal('Maximum response size reached', err && err.message);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should ignore trailing junk

done => {  
 request.get(`${base}/junk`).end((err, res) => {  
 res.should.have.status(200);  
 res.text.should.equal(subject);  
 done();  
 });  
 }

should ignore missing data

done => {  
 request.get(`${base}/chopped`).end((err, res) => {  
 assert.equal(undefined, err);  
 res.should.have.status(200);  
 res.text.should.startWith(subject);  
 done();  
 });  
 }

should handle corrupted responses

done => {  
 request.get(`${base}/corrupt`).end((err, res) => {  
 assert(err, 'missing error');  
 assert(!res, 'response should not be defined');  
 done();  
 });  
 }

should handle no content with gzip header

done => {  
 request.get(`${base}/nocontent`).end((err, res) => {  
 assert.ifError(err);  
 assert(res);  
 res.should.have.status(204);  
 res.text.should.equal('');  
 res.headers.should.not.have.property('content-length');  
 done();  
 });  
 }

# without encoding set

should buffer if asked

return request  
 .get(`${base}/binary`)  
 .buffer(true)  
 .then(res => {  
 res.should.have.status(200);  
 assert(res.headers['content-length']);  
 assert(res.body.byteLength);  
 assert.equal(subject, res.body.toString());  
 });

should emit buffers

done => {  
 request.get(`${base}/binary`).end((err, res) => {  
 res.should.have.status(200);  
 res.headers['content-length'].should.be.below(subject.length);  
 res.on('data', chunk => {  
 chunk.should.have.length(subject.length);  
 });  
 res.on('end', done);  
 });  
 }

# Multipart

# #field(name, value)

should set a multipart field value

const req = request.post(`${base}/echo`);  
req.field('user[name]', 'tobi');  
req.field('user[age]', '2');  
req.field('user[species]', 'ferret');  
return req.then(res => {  
 res.body['user[name]'].should.equal('tobi');  
 res.body['user[age]'].should.equal('2');  
 res.body['user[species]'].should.equal('ferret');  
});

should work with file attachments

const req = request.post(`${base}/echo`);  
req.field('name', 'Tobi');  
req.attach('document', 'test/node/fixtures/user.html');  
req.field('species', 'ferret');  
return req.then(res => {  
 res.body.name.should.equal('Tobi');  
 res.body.species.should.equal('ferret');  
 const html = res.files.document;  
 html.name.should.equal('user.html');  
 html.type.should.equal('text/html');  
 read(html.path).should.equal('<h1>name</h1>');  
});

# #attach(name, path)

should attach a file

const req = request.post(`${base}/echo`);  
req.attach('one', 'test/node/fixtures/user.html');  
req.attach('two', 'test/node/fixtures/user.json');  
req.attach('three', 'test/node/fixtures/user.txt');  
return req.then(res => {  
 const html = res.files.one;  
 const json = res.files.two;  
 const text = res.files.three;  
 html.name.should.equal('user.html');  
 html.type.should.equal('text/html');  
 read(html.path).should.equal('<h1>name</h1>');  
 json.name.should.equal('user.json');  
 json.type.should.equal('application/json');  
 read(json.path).should.equal('{"name":"tobi"}');  
 text.name.should.equal('user.txt');  
 text.type.should.equal('text/plain');  
 read(text.path).should.equal('Tobi');  
});

# when a file does not exist

should fail the request with an error

done => {  
 const req = request.post(`${base}/echo`);  
 req.attach('name', 'foo');  
 req.attach('name2', 'bar');  
 req.attach('name3', 'baz');  
 req.end((err, res) => {  
 assert.ok(Boolean(err), 'Request should have failed.');  
 err.code.should.equal('ENOENT');  
 err.message.should.containEql('ENOENT');  
 err.path.should.equal('foo');  
 done();  
 });  
 }

promise should fail

return request  
 .post(`${base}/echo`)  
 .field({ a: 1, b: 2 })  
 .attach('c', 'does-not-exist.txt')  
 .then(  
 res => assert.fail('It should not allow this'),  
 err => {  
 err.code.should.equal('ENOENT');  
 err.path.should.equal('does-not-exist.txt');  
 }  
 );

should report ECONNREFUSED via the callback

done => {  
 request  
 .post('http://127.0.0.1:1') // nobody is listening there  
 .attach('name', 'file-does-not-exist')  
 .end((err, res) => {  
 assert.ok(Boolean(err), 'Request should have failed');  
 err.code.should.equal('ECONNREFUSED');  
 done();  
 });  
 }

should report ECONNREFUSED via Promise

return request  
 .post('http://127.0.0.1:1') // nobody is listening there  
 .attach('name', 'file-does-not-exist')  
 .then(  
 res => assert.fail('Request should have failed'),  
 err => err.code.should.equal('ECONNREFUSED')  
 );

# #attach(name, path, filename)

should use the custom filename

request  
 .post(`${base}/echo`)  
 .attach('document', 'test/node/fixtures/user.html', 'doc.html')  
 .then(res => {  
 const html = res.files.document;  
 html.name.should.equal('doc.html');  
 html.type.should.equal('text/html');  
 read(html.path).should.equal('<h1>name</h1>');  
 })

should fire progress event

done => {  
 let loaded = 0;  
 let total = 0;  
 let uploadEventWasFired = false;  
 request  
 .post(`${base}/echo`)  
 .attach('document', 'test/node/fixtures/user.html')  
 .on('progress', event => {  
 total = event.total;  
 loaded = event.loaded;  
 if (event.direction === 'upload') {  
 uploadEventWasFired = true;  
 }  
 })  
 .end((err, res) => {  
 if (err) return done(err);  
 const html = res.files.document;  
 html.name.should.equal('user.html');  
 html.type.should.equal('text/html');  
 read(html.path).should.equal('<h1>name</h1>');  
 total.should.equal(223);  
 loaded.should.equal(223);  
 uploadEventWasFired.should.equal(true);  
 done();  
 });  
 }

filesystem errors should be caught

done => {  
 request  
 .post(`${base}/echo`)  
 .attach('filedata', 'test/node/fixtures/non-existent-file.ext')  
 .end((err, res) => {  
 assert.ok(Boolean(err), 'Request should have failed.');  
 err.code.should.equal('ENOENT');  
 err.path.should.equal('test/node/fixtures/non-existent-file.ext');  
 done();  
 });  
 }

# #field(name, val)

should set a multipart field value

done => {  
 request  
 .post(`${base}/echo`)  
 .field('first-name', 'foo')  
 .field('last-name', 'bar')  
 .end((err, res) => {  
 if (err) done(err);  
 res.should.be.ok();  
 res.body['first-name'].should.equal('foo');  
 res.body['last-name'].should.equal('bar');  
 done();  
 });  
 }

# #field(object)

should set multiple multipart fields

done => {  
 request  
 .post(`${base}/echo`)  
 .field({ 'first-name': 'foo', 'last-name': 'bar' })  
 .end((err, res) => {  
 if (err) done(err);  
 res.should.be.ok();  
 res.body['first-name'].should.equal('foo');  
 res.body['last-name'].should.equal('bar');  
 done();  
 });  
 }

# with network error

should error

request.get(`http://localhost:${this.port}/`).end((err, res) => {  
 assert(err, 'expected an error');  
 done();  
});

# request

# not modified

should start with 200

done => {  
 request.get(`${base}/if-mod`).end((err, res) => {  
 res.should.have.status(200);  
 res.text.should.match(/^\d+$/);  
 ts = Number(res.text);  
 done();  
 });  
 }

should then be 304

done => {  
 request  
 .get(`${base}/if-mod`)  
 .set('If-Modified-Since', new Date(ts).toUTCString())  
 .end((err, res) => {  
 res.should.have.status(304);  
 // res.text.should.be.empty  
 done();  
 });  
 }

# req.parse(fn)

should take precedence over default parsers

done => {  
 request  
 .get(`${base}/manny`)  
 .parse(request.parse['application/json'])  
 .end((err, res) => {  
 assert(res.ok);  
 assert.equal('{"name":"manny"}', res.text);  
 assert.equal('manny', res.body.name);  
 done();  
 });  
 }

should be the only parser

request  
 .get(`${base}/image`)  
 .buffer(false)  
 .parse((res, fn) => {  
 res.on('data', () => {});  
 })  
 .then(res => {  
 assert(res.ok);  
 assert.strictEqual(res.text, undefined);  
 res.body.should.eql({});  
 })

should emit error if parser throws

done => {  
 request  
 .get(`${base}/manny`)  
 .parse(() => {  
 throw new Error('I am broken');  
 })  
 .on('error', err => {  
 err.message.should.equal('I am broken');  
 done();  
 })  
 .end();  
 }

should emit error if parser returns an error

done => {  
 request  
 .get(`${base}/manny`)  
 .parse((res, fn) => {  
 fn(new Error('I am broken'));  
 })  
 .on('error', err => {  
 err.message.should.equal('I am broken');  
 done();  
 })  
 .end();  
 }

should not emit error on chunked json

done => {  
 request.get(`${base}/chunked-json`).end(err => {  
 assert.ifError(err);  
 done();  
 });  
 }

should not emit error on aborted chunked json

done => {  
 const req = request.get(`${base}/chunked-json`);  
 req.end(err => {  
 assert.ifError(err);  
 done();  
 });  
 setTimeout(() => {  
 req.abort();  
 }, 50);  
 }

# pipe on redirect

should follow Location

done => {  
 const stream = fs.createWriteStream(destPath);  
 const redirects = [];  
 const req = request  
 .get(base)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .connect({  
 inapplicable: 'should be ignored'  
 });  
 stream.on('finish', () => {  
 redirects.should.eql(['/movies', '/movies/all', '/movies/all/0']);  
 fs.readFileSync(destPath, 'utf8').should.eql('first movie page');  
 done();  
 });  
 req.pipe(stream);  
 }

# request pipe

should act as a writable stream

done => {  
 const req = request.post(base);  
 const stream = fs.createReadStream('test/node/fixtures/user.json');  
 req.type('json');  
 req.on('response', res => {  
 res.body.should.eql({ name: 'tobi' });  
 done();  
 });  
 stream.pipe(req);  
 }

end() stops piping

done => {  
 const stream = fs.createWriteStream(destPath);  
 request.get(base).end((err, res) => {  
 try {  
 res.pipe(stream);  
 return done(new Error('Did not prevent nonsense pipe'));  
 } catch (err2) {  
 /\* expected error \*/  
 }  
 done();  
 });  
 }

should act as a readable stream

done => {  
 const stream = fs.createWriteStream(destPath);  
 let responseCalled = false;  
 const req = request.get(base);  
 req.type('json');  
 req.on('response', res => {  
 res.status.should.eql(200);  
 responseCalled = true;  
 });  
 stream.on('finish', () => {  
 JSON.parse(fs.readFileSync(destPath, 'utf8')).should.eql({  
 name: 'tobi'  
 });  
 responseCalled.should.be.true();  
 done();  
 });  
 req.pipe(stream);  
 }

# req.query(String)

should support passing in a string

done => {  
 request  
 .del(base)  
 .query('name=t%F6bi')  
 .end((err, res) => {  
 res.body.should.eql({ name: 't%F6bi' });  
 done();  
 });  
 }

should work with url query-string and string for query

done => {  
 request  
 .del(`${base}/?name=tobi`)  
 .query('age=2%20')  
 .end((err, res) => {  
 res.body.should.eql({ name: 'tobi', age: '2 ' });  
 done();  
 });  
 }

should support compound elements in a string

done => {  
 request  
 .del(base)  
 .query('name=t%F6bi&age=2')  
 .end((err, res) => {  
 res.body.should.eql({ name: 't%F6bi', age: '2' });  
 done();  
 });  
 }

should work when called multiple times with a string

done => {  
 request  
 .del(base)  
 .query('name=t%F6bi')  
 .query('age=2%F6')  
 .end((err, res) => {  
 res.body.should.eql({ name: 't%F6bi', age: '2%F6' });  
 done();  
 });  
 }

should work with normal `query` object and query string

done => {  
 request  
 .del(base)  
 .query('name=t%F6bi')  
 .query({ age: '2' })  
 .end((err, res) => {  
 res.body.should.eql({ name: 't%F6bi', age: '2' });  
 done();  
 });  
 }

should not encode raw backticks, but leave encoded ones as is

return Promise.all([  
 request  
 .get(`${base}/raw-query`)  
 .query('name=`t%60bi`&age`=2')  
 .then(res => {  
 res.text.should.eql('name=`t%60bi`&age`=2');  
 }),  
 request.get(base + '/raw-query?`age%60`=2%60`').then(res => {  
 res.text.should.eql('`age%60`=2%60`');  
 }),  
 request  
 .get(`${base}/raw-query`)  
 .query('name=`t%60bi`')  
 .query('age`=2')  
 .then(res => {  
 res.text.should.eql('name=`t%60bi`&age`=2');  
 })  
]);

# req.query(Object)

should construct the query-string

done => {  
 request  
 .del(base)  
 .query({ name: 'tobi' })  
 .query({ order: 'asc' })  
 .query({ limit: ['1', '2'] })  
 .end((err, res) => {  
 res.body.should.eql({ name: 'tobi', order: 'asc', limit: ['1', '2'] });  
 done();  
 });  
 }

should encode raw backticks

done => {  
 request  
 .get(`${base}/raw-query`)  
 .query({ name: '`tobi`' })  
 .query({ 'orde%60r': null })  
 .query({ '`limit`': ['%602`'] })  
 .end((err, res) => {  
 res.text.should.eql('name=%60tobi%60&orde%2560r&%60limit%60=%25602%60');  
 done();  
 });  
 }

should not error on dates

done => {  
 const date = new Date(0);  
 request  
 .del(base)  
 .query({ at: date })  
 .end((err, res) => {  
 assert.equal(date.toISOString(), res.body.at);  
 done();  
 });  
 }

should work after setting header fields

done => {  
 request  
 .del(base)  
 .set('Foo', 'bar')  
 .set('Bar', 'baz')  
 .query({ name: 'tobi' })  
 .query({ order: 'asc' })  
 .query({ limit: ['1', '2'] })  
 .end((err, res) => {  
 res.body.should.eql({ name: 'tobi', order: 'asc', limit: ['1', '2'] });  
 done();  
 });  
 }

should append to the original query-string

done => {  
 request  
 .del(`${base}/?name=tobi`)  
 .query({ order: 'asc' })  
 .end((err, res) => {  
 res.body.should.eql({ name: 'tobi', order: 'asc' });  
 done();  
 });  
 }

should retain the original query-string

done => {  
 request.del(`${base}/?name=tobi`).end((err, res) => {  
 res.body.should.eql({ name: 'tobi' });  
 done();  
 });  
 }

should keep only keys with null querystring values

done => {  
 request  
 .del(`${base}/url`)  
 .query({ nil: null })  
 .end((err, res) => {  
 res.text.should.equal('/url?nil');  
 done();  
 });  
 }

query-string should be sent on pipe

done => {  
 const req = request.put(`${base}/?name=tobi`);  
 const stream = fs.createReadStream('test/node/fixtures/user.json');  
 req.on('response', res => {  
 res.body.should.eql({ name: 'tobi' });  
 done();  
 });  
 stream.pipe(req);  
 }

# request.get

# on 301 redirect

should follow Location with a GET request

done => {  
 const req = request.get(`${base}/test-301`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# on 302 redirect

should follow Location with a GET request

done => {  
 const req = request.get(`${base}/test-302`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# on 303 redirect

should follow Location with a GET request

done => {  
 const req = request.get(`${base}/test-303`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# on 307 redirect

should follow Location with a GET request

done => {  
 const req = request.get(`${base}/test-307`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# on 308 redirect

should follow Location with a GET request

done => {  
 const req = request.get(`${base}/test-308`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# request.post

# on 301 redirect

should follow Location with a GET request

done => {  
 const req = request.post(`${base}/test-301`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# on 302 redirect

should follow Location with a GET request

done => {  
 const req = request.post(`${base}/test-302`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# on 303 redirect

should follow Location with a GET request

done => {  
 const req = request.post(`${base}/test-303`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('GET');  
 done();  
 });  
 }

# on 307 redirect

should follow Location with a POST request

done => {  
 const req = request.post(`${base}/test-307`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('POST');  
 done();  
 });  
 }

# on 308 redirect

should follow Location with a POST request

done => {  
 const req = request.post(`${base}/test-308`).redirects(1);  
 req.end((err, res) => {  
 req.req.\_headers.host.should.eql(`localhost:${server2.address().port}`);  
 res.status.should.eql(200);  
 res.text.should.eql('POST');  
 done();  
 });  
 }

# request

# on redirect

should merge cookies if agent is used

done => {  
 request  
 .agent()  
 .get(`${base}/cookie-redirect`)  
 .set('Cookie', 'orig=1; replaced=not')  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(/orig=1/.test(res.text), 'orig=1/.test');  
 assert(/replaced=yes/.test(res.text), 'replaced=yes/.test');  
 assert(/from-redir=1/.test(res.text), 'from-redir=1');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should not merge cookies if agent is not used

done => {  
 request  
 .get(`${base}/cookie-redirect`)  
 .set('Cookie', 'orig=1; replaced=not')  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(/orig=1/.test(res.text), '/orig=1');  
 assert(/replaced=not/.test(res.text), '/replaced=not');  
 assert(!/replaced=yes/.test(res.text), '!/replaced=yes');  
 assert(!/from-redir/.test(res.text), '!/from-redir');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should have previously set cookie for subsquent requests when agent is used

done => {  
 const agent = request.agent();  
 agent.get(`${base}/set-cookie`).end(err => {  
 assert.ifError(err);  
 agent  
 .get(`${base}/show-cookies`)  
 .set({ Cookie: 'orig=1' })  
 .end((err, res) => {  
 try {  
 assert.ifError(err);  
 assert(/orig=1/.test(res.text), 'orig=1/.test');  
 assert(/persist=123/.test(res.text), 'persist=123');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 });  
 }

should follow Location

done => {  
 const redirects = [];  
 request  
 .get(base)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .end((err, res) => {  
 try {  
 const arr = ['/movies', '/movies/all', '/movies/all/0'];  
 redirects.should.eql(arr);  
 res.text.should.equal('first movie page');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should follow Location with IP override

const redirects = [];  
const url = URL.parse(base);  
return request  
 .get(`http://redir.example.com:${url.port || '80'}${url.pathname}`)  
 .connect({  
 '\*': url.hostname  
 })  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .then(res => {  
 const arr = ['/movies', '/movies/all', '/movies/all/0'];  
 redirects.should.eql(arr);  
 res.text.should.equal('first movie page');  
 });

should not follow on HEAD by default

const redirects = [];  
return request  
 .head(base)  
 .ok(() => true)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .then(res => {  
 redirects.should.eql([]);  
 res.status.should.equal(302);  
 });

should follow on HEAD when redirects are set

done => {  
 const redirects = [];  
 request  
 .head(base)  
 .redirects(10)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .end((err, res) => {  
 try {  
 const arr = [];  
 arr.push('/movies');  
 arr.push('/movies/all');  
 arr.push('/movies/all/0');  
 redirects.should.eql(arr);  
 assert(!res.text);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should remove Content-\* fields

done => {  
 request  
 .post(`${base}/header`)  
 .type('txt')  
 .set('X-Foo', 'bar')  
 .set('X-Bar', 'baz')  
 .send('hey')  
 .end((err, res) => {  
 try {  
 assert(res.body);  
 res.body.should.have.property('x-foo', 'bar');  
 res.body.should.have.property('x-bar', 'baz');  
 res.body.should.not.have.property('content-type');  
 res.body.should.not.have.property('content-length');  
 res.body.should.not.have.property('transfer-encoding');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should retain cookies

done => {  
 request  
 .get(`${base}/header`)  
 .set('Cookie', 'foo=bar;')  
 .end((err, res) => {  
 try {  
 assert(res.body);  
 res.body.should.have.property('cookie', 'foo=bar;');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should not resend query parameters

done => {  
 const redirects = [];  
 const query = [];  
 request  
 .get(`${base}/?foo=bar`)  
 .on('redirect', res => {  
 query.push(res.headers.query);  
 redirects.push(res.headers.location);  
 })  
 .end((err, res) => {  
 try {  
 const arr = [];  
 arr.push('/movies');  
 arr.push('/movies/all');  
 arr.push('/movies/all/0');  
 redirects.should.eql(arr);  
 res.text.should.equal('first movie page');  
 query.should.eql(['{"foo":"bar"}', '{}', '{}']);  
 res.headers.query.should.eql('{}');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should handle no location header

done => {  
 request.get(`${base}/bad-redirect`).end((err, res) => {  
 try {  
 err.message.should.equal('No location header for redirect');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# when relative

should redirect to a sibling path

done => {  
 const redirects = [];  
 request  
 .get(`${base}/relative`)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .end((err, res) => {  
 try {  
 redirects.should.eql(['tobi']);  
 res.text.should.equal('tobi');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

should redirect to a parent path

done => {  
 const redirects = [];  
 request  
 .get(`${base}/relative/sub`)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .end((err, res) => {  
 try {  
 redirects.should.eql(['../tobi']);  
 res.text.should.equal('tobi');  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# req.redirects(n)

should alter the default number of redirects to follow

done => {  
 const redirects = [];  
 request  
 .get(base)  
 .redirects(2)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .end((err, res) => {  
 try {  
 const arr = [];  
 assert(res.redirect, 'res.redirect');  
 arr.push('/movies');  
 arr.push('/movies/all');  
 redirects.should.eql(arr);  
 res.text.should.match(/Moved Temporarily|Found/);  
 done();  
 } catch (err2) {  
 done(err2);  
 }  
 });  
 }

# on POST

should redirect as GET

const redirects = [];  
return request  
 .post(`${base}/movie`)  
 .send({ name: 'Tobi' })  
 .redirects(2)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .then(res => {  
 redirects.should.eql(['/movies/all/0']);  
 res.text.should.equal('first movie page');  
 });

using multipart/form-data should redirect as GET

const redirects = [];  
request  
 .post(`${base}/movie`)  
 .type('form')  
 .field('name', 'Tobi')  
 .redirects(2)  
 .on('redirect', res => {  
 redirects.push(res.headers.location);  
 })  
 .then(res => {  
 redirects.should.eql(['/movies/all/0']);  
 res.text.should.equal('first movie page');  
 });

# response

should act as a readable stream

done => {  
 const req = request.get(base).buffer(false);  
 req.end((err, res) => {  
 if (err) return done(err);  
 let trackEndEvent = 0;  
 let trackCloseEvent = 0;  
 res.on('end', () => {  
 trackEndEvent++;  
 trackEndEvent.should.equal(1);  
 if (!process.env.HTTP2\_TEST) {  
 trackCloseEvent.should.equal(0); // close should not have been called  
 }  
 done();  
 });  
 res.on('close', () => {  
 trackCloseEvent++;  
 });  
 (() => {  
 res.pause();  
 }).should.not.throw();  
 (() => {  
 res.resume();  
 }).should.not.throw();  
 (() => {  
 res.destroy();  
 }).should.not.throw();  
 });  
 }

# req.serialize(fn)

should take precedence over default parsers

done => {  
 request  
 .post(`${base}/echo`)  
 .send({ foo: 123 })  
 .serialize(data => '{"bar":456}')  
 .end((err, res) => {  
 assert.ifError(err);  
 assert.equal('{"bar":456}', res.text);  
 assert.equal(456, res.body.bar);  
 done();  
 });  
 }

# request.get().set()

should set host header after get()

done => {  
 app.get('/', (req, res) => {  
 assert.equal(req.hostname, 'example.com');  
 res.end();  
 });  
 server = http.createServer(app);  
 server.listen(0, function listening() {  
 request  
 .get(`http://localhost:${server.address().port}`)  
 .set('host', 'example.com')  
 .then(() => {  
 return request  
 .get(`http://example.com:${server.address().port}`)  
 .connect({  
 'example.com': 'localhost',  
 '\*': 'fail'  
 });  
 })  
 .then(() => done(), done);  
 });  
 }

# res.toError()

should return an Error

done => {  
 request.get(base).end((err, res) => {  
 var err = res.toError();  
 assert.equal(err.status, 400);  
 assert.equal(err.method, 'GET');  
 assert.equal(err.path, '/');  
 assert.equal(err.message, 'cannot GET / (400)');  
 assert.equal(err.text, 'invalid json');  
 done();  
 });  
 }

# [unix-sockets] http

# request

path: / (root)

done => {  
 request.get(`${base}/`).end((err, res) => {  
 assert(res.ok);  
 assert.strictEqual('root ok!', res.text);  
 done();  
 });  
 }

path: /request/path

done => {  
 request.get(`${base}/request/path`).end((err, res) => {  
 assert(res.ok);  
 assert.strictEqual('request path ok!', res.text);  
 done();  
 });  
 }

# [unix-sockets] https

# request

path: / (root)

done => {  
 request  
 .get(`${base}/`)  
 .ca(cacert)  
 .end((err, res) => {  
 assert.ifError(err);  
 assert(res.ok);  
 assert.strictEqual('root ok!', res.text);  
 done();  
 });  
 }

path: /request/path

done => {  
 request  
 .get(`${base}/request/path`)  
 .ca(cacert)  
 .end((err, res) => {  
 assert.ifError(err);  
 assert(res.ok);  
 assert.strictEqual('request path ok!', res.text);  
 done();  
 });  
 }

# req.get()

should set a default user-agent

request.get(`${base}/ua`).then(res => {  
 assert(res.headers);  
 assert(res.headers['user-agent']);  
 assert(  
 /^node-superagent\/\d+\.\d+\.\d+(?:-[a-z]+\.\d+|$)/.test(  
 res.headers['user-agent']  
 )  
 );  
 })

should be able to override user-agent

request  
 .get(`${base}/ua`)  
 .set('User-Agent', 'foo/bar')  
 .then(res => {  
 assert(res.headers);  
 assert.equal(res.headers['user-agent'], 'foo/bar');  
 })

should be able to wipe user-agent

request  
 .get(`${base}/ua`)  
 .unset('User-Agent')  
 .then(res => {  
 assert(res.headers);  
 assert.equal(res.headers['user-agent'], void 0);  
 })

# utils.type(str)

should return the mime type

utils  
 .type('application/json; charset=utf-8')  
 .should.equal('application/json');  
utils.type('application/json').should.equal('application/json');

# utils.params(str)

should return the field parameters

const obj = utils.params('application/json; charset=utf-8; foo = bar');  
obj.charset.should.equal('utf-8');  
obj.foo.should.equal('bar');  
utils.params('application/json').should.eql({});

# utils.parseLinks(str)

should parse links

const str =  
 '<https://api.github.com/repos/visionmedia/mocha/issues?page=2>; rel="next", <https://api.github.com/repos/visionmedia/mocha/issues?page=5>; rel="last"';  
const ret = utils.parseLinks(str);  
ret.next.should.equal(  
 'https://api.github.com/repos/visionmedia/mocha/issues?page=2'  
);  
ret.last.should.equal(  
 'https://api.github.com/repos/visionmedia/mocha/issues?page=5'  
);

