dnsimple

Rust API Cheatsheet

With the DNSimple crate you can easily interact our powerful API to administer domain names, configure DNS records, provision and install SSL certificates, and more.

Getting Started

1. Install the Rust crate

`cargo install dnsimple`

2. Authenticate

Obtain your API access token: https://support.dnsimple.com/articles/api-access-token/

```
use dnsimple::dnsimple::{Client, new_client};
let client = new_client(true, String::from("AUTH_TOKEN"));
```

3. Check Authorization

If you want to know which account is associated with the current access token, you can use `#identity`. The account ID is required for the majority of API operations.

```
let whoami = client
    .identity()
    .whoami()
    .unwrap()
    .data
    .unwrap();

let account = whoami.account.unwrap();
let account_id = accound.id;

println!(
  "{} (your account ID)",
  account_id
);
=> 1234 (your account ID)
```

Managing Domains

Check Domain Availability

Check if a domain is available for registration.

```
let domain_check = client
    .registrar()
    .check_domain(account_id, "foo.com")
    .unwrap()
    .data
    .unwrap();

println!(
  "Domain: {}\nAvailable: {}\nPremium: {}",
  domain_check.domain, domain_check.available, domain_check.premium
);
=> Domain: foo.com
Available: true
Premium: false
```

Register A Domain

1. To register a domain, you need to specify a registrant_id. This can be fetched via the Contacts API.

```
let contacts = client
    .contacts()
    .list_contacts(account_id, None)
    .unwrap()
    .data
    .unwrap();

let first_contact = contacts.first.unwrap();

println!(
    "{}"
    ,first_contact.id
);
=> 123
```

2. You can register the domain with this information.

```
let domain_registration_payload = DomainRegistrationPayload {
    registrant_id: first_contact.id,
    whois_privacy: None,
    auto_renew: None,
    extended_attributes: None,
    premium_price: None,
};

let domain_registration = client
    .registrar()
    .register_domain(account_id, "foo.com", domain_registration_payload)
    .unwrap()
    .data
    .unwrap();

println!(
  "State: {}\nAuto Renew: {}\nWhois Privacy: {}\nRegistrant:{}",
  domain_registration.state,
  domain_registration.auto_renew,
  domain_registration.whois_privacy,
  domain_registration.registrant_id
);

=>State: registered
Auto Renew: false
Whois Privacy: false
Registrant: 123
```

DNS

Create a DNS record

Create a DNS A record to map an IP address to a domain.

```
let zone_record_payload = ZoneRecordPayload {
    name: String::from("www"),
    record_type: String::from("A"),
    content: String::from("127.0.0.1"),
    ttl: None,
    priority: None,
    regions: None,

};

let record = client
    .zones()
    .create_zone_record(account_id, "foo.com", zone_record_payload)
    .unwrap()
    .data
    .unwrap();

println!(
  "ID: {}\nZone: {}\nName: {}\nType: {}\nContent: {}",
  record.id, record.zone_id, record.type, record.content
);

=>ID: 123
Zone: foo.com
Name: www
Type: A
Content: 137.0.0.1
```

Update a DNS record

Update a previously created DNS record.

```
let update = ZoneRecordUpdatePayload {
    name: None,
    content: None,
    ttl: Option::from(60),
    priority: None,
    regions: None,
};

let updated_zone_record = client
    .zones()
    .update_zone_record(account_id, "foo.com", record.id, update)
    .unwrap()
    .data
    .unwrap();

println!(
  "ID: {}\nUpdated TTL: {}",
  updated_zone_record.id, updated_zone_record.ttl
);

=>ID: 123
Updated TTL: 60
```

SSL Certificates

Order an SSL Certificate with Let's Encrypt

Creates the purchase order. Use the ID to issue the certificate.

```
let payload = LetsEncryptPurchasePayload {
    auto_renew: false,
    name: String::from("test-certificate"),
    alternate_names: vec![],
};

let cert = client
    .certificates()
    .purchase_letsencrypt_certificate(account_id, "foo.com", payload)
    .unwrap()
    .data
    .unwrap();

println!(
  "ID: {}\nState: {}",
  cert.id, cert.state
);

=>ID: 123
State: new
```

Issue an Let's Encrypt Certificate

Issues the pending order. This process is async. A successful response means that the response is queued.

```
let cert_issue = client
    .certificates()
    .issue_letsencrypt_certificate(account_id, "foo.com", cert.id)
    .unwrap()
    .data
    .unwrap();

println!(
  "State: {}",
  cert_issue.state
);

=>State: requesting
```

Install the certificate

Download the certificate.

```
let certificate = client
    .certificates.download_certificate(account_id, "foo.com", cert.id)
    .unwrap()
    .data
    .unwrap();


let mut chain = "".to_string();
for value in certificate.chain {
  chain.push_str(value);
}

let mut file = File::create("www_foo_com.pem")?;
let file_content = format!("{}\n{}\n, certificate.server, chain);
file.write_all(file_content);
```

Download the certficate's private key.

```
let key = client
    .certificates()
    .certificate_private_key(account_id, "foo.com", certificate.id)
    .unwrap()
    .data
    .unwrap();

fs::write("www_foo_com.key", key.private_key);
```