

---

# Rel Documentation

*Release v0.10.1*

**ChromaWay AB**

**Sep 23, 2022**



---

## Contents

---

<b>1</b>	<b>Rell language</b>	<b>3</b>
<b>2</b>	<b>Chromia</b>	<b>5</b>
2.1	Get Started with Web IDE . . . . .	5
2.2	Rell Basics . . . . .	9
2.3	Eclipse IDE . . . . .	18
2.4	Example Projects . . . . .	47
2.5	Language Features . . . . .	60
2.6	Advanced Topics . . . . .	103
2.7	Run.XML . . . . .	150
2.8	Testing module . . . . .	154
2.9	Upgrading to Rell 0.10.x . . . . .	161



This section will cover the advantages of Rell and its position within the Chromia platform.

If you are eager to get started with Rell, you can safely skip straight to the [Quick Start](#) section.

For code references, visit [Language Features](#).



# CHAPTER 1

---

## Rell language

---

Most dapp blockchain platforms use virtual machines of various kinds. But a traditional virtual machine architecture doesn't work very well with the Chromia relational data model, as we need a way to encode queries as well as operations. For this reason, we are taking a more language-centric approach: a new language called Rell (Relational language) will be used for dapp programming. This language allows programmers to describe the data model/schema, queries, and procedural application code.

Rell code is compiled to an intermediate binary format which can be understood as code for a specialized virtual machine. Chromia nodes will then translate queries contained in this code into SQL (while making sure this translation is safe) and execute code as needed using an interpreter or compiler.

Rell has the following features:

- Type safety / static type checks. It's very important to catch programming errors at the compilation stage to prevent financial losses. Rell is much more type-safe than SQL, and it makes sure that types returned by queries match types used in procedural code.
- Safety-optimized. Arithmetic operations are safe right out of the box, programmers do not need to worry about overflows. Authorization checks are explicitly required.
- Concise, expressive and convenient. Many developers dislike SQL because it is highly verbose. Rell doesn't bother developers with details which can be derived automatically. As a data definition language, Rell is up to 7x more compact than SQL.
- Allows meta-programming. We do not want application developers to implement the basics from scratch for every dapp. Rell will allow functionality to be bundled as templates.

Our research indicated that no existing language or environment has this feature set, and thus development of a new language was absolutely necessary.

We designed Rell in such a way that it is easy to learn for programmers:

- Programmers can use relational programming idioms they are already familiar with. However, they don't have to go out of their way to express everything through relational algebra: Rell can seamlessly merge relational constructs with procedural programming.
- The language is deliberately similar to modern programming languages like JavaScript and Kotlin. A familiar language is easier to adapt to, and our internal tests show that programmers can become proficient in Rell

in a matter of days. In contrast, the ALGOL-style syntax of PL/SQL generally feels unintuitive to modern developers.



Rel is built for [Chromia](#). Chromia is a new blockchain platform for decentralized applications, conceived in response to the shortcomings of existing platforms and designed to enable a new generation of dapps to scale beyond what is currently possible

While platforms such as Ethereum allow any kind of application to be implemented in theory, in practice they have many limitations: bad user experience, high fees, frustrating developer experience, poor security. This prevents decentralized apps (dapps) from going mainstream.

We believe that to address these problems properly we need to seriously rethink the blockchain architecture and programming model with the needs of decentralized applications in mind. Our priorities are to:

- Allow dapps to scale to millions of users.
- Improve the user experience of dapps to achieve parity with centralized applications.
- Allow developers to build secure applications with using familiar paradigms.

## 2.1 Get Started with Web IDE

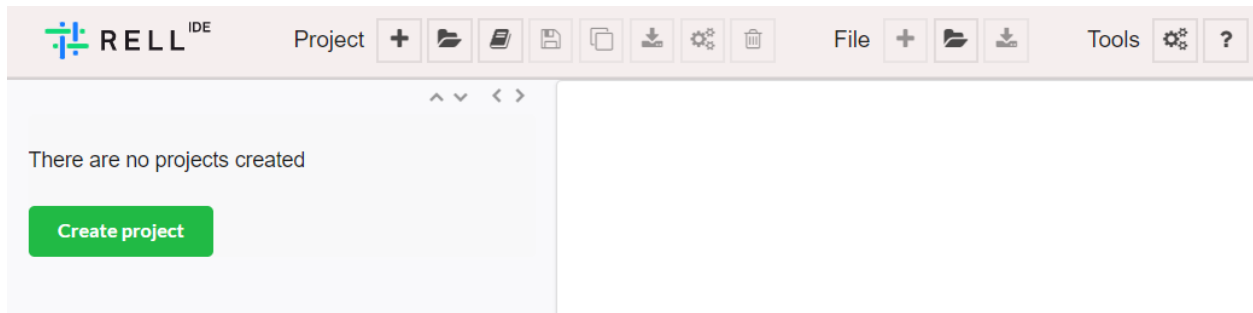
---

**Important:** Rel Web IDE is available at <https://rellide-staging.chromia.dev/>

---

Upon entering, you will see an interface similar to the image below.

Click **Create Project** button:



This will open a modal element where you can specify the name of the Module and the Language used (in this example: Rel).

For convenience you can include template and test code.

## Create project

### Project name

Set project name

Project name must begin with latin letter and contain only letters and numbers

### Project type

Rel

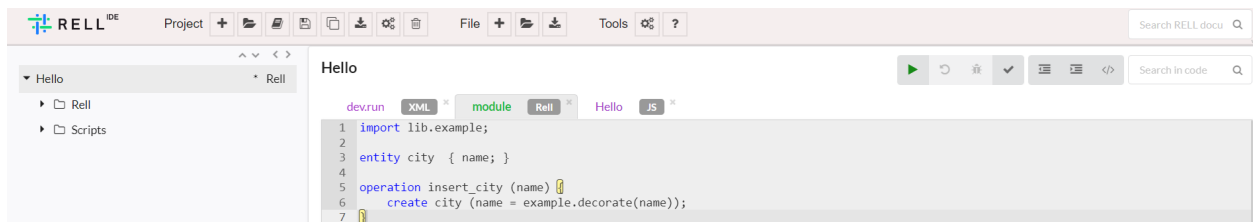
### Code template

Hello world

Create

Close

Click on the “Create” button. The screen will now be filled with code.



**Browser** To the left bar is the Browser. You can use it to work with several examples.

**Editor** Inside the central element on the right the editor filled with a template of source code.

**Buttons** On top of the Editor there is a button to “Start Node” (the green “Play” icon), don’t press that one yet. There is also a button “Run tests” (the gray “bug” icon).

### 2.1.1 Hello World!

As a minimal first application, you can make a Hello World example with a focus on Ukraine.

If you checked the *use template* box and look at editor section on the top, you will see this code as a template:

```
entity city { key name; }

operation insert_city (name) {
  create city (name);
}
```

This is a small registry of cities.

Don’t worry about the detail of this code yet, we will come to them in a bit. For now, let’s confirm that our code template is working properly.

In order to run the code we need a test in javascript. If you switch to the `Hello.js` test file, you will see it’s filled with some test written in javascript:

```
const tx = gtx.newTransaction([user.pubKey]);

tx.addOperation('insert_city', "Kiev");

tx.sign(user.privKey, user.pubKey);

return tx.postAndWaitConfirmation();
```

Click the ‘Run tests’ button, and a green message will appear.

1 `const tx = gtx.newTransaction([user.pubKey]);`  
 2  
 3 `tx.addOperation('insert_city', "Kiev");`  
 4  
 5 `tx.sign(user.privKey, user.pubKey);`  
 6  
 7 `return tx.postAndWaitConfirmation();`  
 8

Session JS tests XML tests Log \* Console ►

✓ Tests passed ↻

Output:  
null

Congratulations! After all this work, we suggest that you put “Relational Blockchain” on your CV.

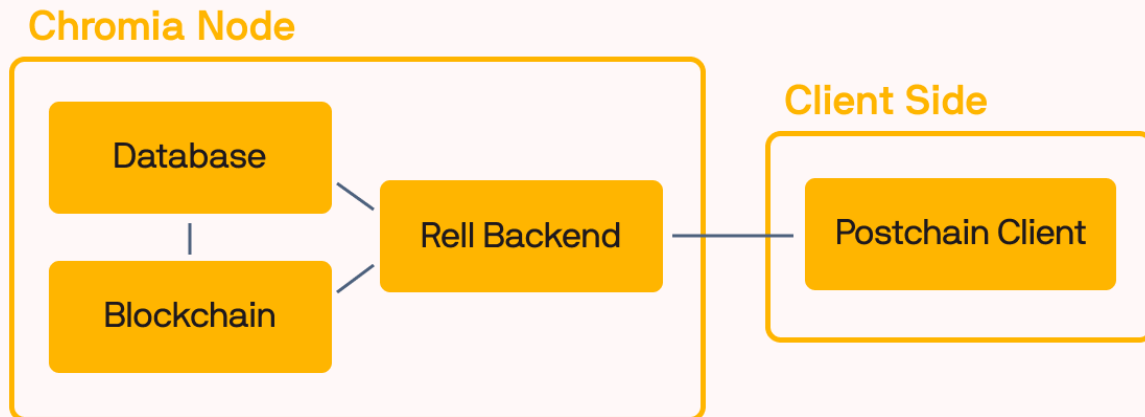
## 2.1.2 Where to go next?

Next step is to learn about what the Rel code actually means in the [Rel Basics](#) section.

But if you prefer learning by example, you can choose to start with one of our [Example Projects](#) instead.

## 2.2 Rell Basics

This chapter will cover basic Rell syntax and functionality, as well as how we write our client side so that it can communicate with our blockchain. Rell is a language designed for relational blockchain programming. The structure of a decentralized application built in Rell will look something like this:



The end user will be communicating with our client side, which in turn will send transactions to Rell using a postchain client. There are currently postchain clients for JavaScript, C# and Java, and we will be using JavaScript for our client side example.

- The *Main Concepts* section guides you through the concepts needed to create a program in Rell.
- While *Client Side* describes how to work with a Rell backend using a JavaScript client.

### 2.2.1 Main Concepts

#### Language overview

Rell is a language for relational blockchain programming. It combines the following features:

1. Relational data modeling and queries similar to SQL. People familiar with SQL should feel at home once they learn the new syntax.
2. Normal programming constructs: variables, loops, functions, collections, etc.
3. Constructs which specifically target application backends and, in particular, blockchain-style programming including request routing, authorization, etc.

Rell aims to make programming as ergonomic as possible. It minimizes boilerplate and repetition. At the same time, as a static type system it can detect and prevent many kinds of defects.

#### Blockchain programming

There are many different styles of blockchain programming. In the context of Rell, we see blockchain as a method for secure synchronization of databases on nodes of the system. Thus Rell is very database-centric.

Programming in Rell is pretty much identical to programming application backends: you need to handle requests to modify the data in the database and other requests which retrieve data from a database. Handling these two types

of requests is basically all that a backend does. But, of course, before you implement request handlers, you need to describe your data model first.

## Entity definitions

In SQL, usually you define your data model using `CREATE TABLE` syntax. In Java, you can define data objects using `class` definition. In Rell, we define them as `entity`.

Rell uses persistent objects, thus an entity definition automatically creates the storage (e.g. a table) necessary to persist objects of an entity. As you might expect, Rell's entity definition includes a list of attributes:

```
entity user {
    pubkey: pubkey;
    name: text;
}
```

It is very common that the name of the attribute is the same as its type. For example, it makes sense to call user's pubkey "pubkey." Rell allows you to shorten `pubkey: pubkey;` to just `pubkey;`. Rell also has a number of convenient semantic types, so there is a type called `name` as well. Thus you can rewrite the definition above as just:

```
entity user { pubkey; name; }
```

Typically a system should not allow different users to have the same name. That is, names should be unique. If name is unique, it can be used to identify a user. In Rell, this can be done by defining a `key`, i.e. `key name;`. Note that it's not necessary to define both `key` and attribute. Rell is smart enough to figure out that if you use an attribute in a `key`, that attribute should exist in an entity.

It also might be useful to find a user by his pubkey. Should it also be unique? Not necessarily. A user might have several different identities. When you want to enable fast retrieval, but do not need uniqueness, you can use `index` definition:

```
entity user {
    key name;
    index pubkey;
}
```

However, if you want pubkey to be unique for a user, you can add a second `key`:

```
entity user {
    key name;
    key pubkey;
}
```

Typically, when you define a class in a programming language, it creates a type which can be used to refer to instances of that class. This is exactly how it works in Rell. The definition of entity `user` creates a type `user` which is a type of references to objects stored in a database. References can themselves be used as attributes. For example, you might want to define something owned by a user, say, a channel. You can describe it like this:

```
entity channel {
    index owner: user;
    key name;
}
```

`index` makes it possible to efficiently find all channels owned by a user. `key` makes sure that channel names are unique within the system.

Let's analyze `channel` entity definition from a point of view of a traditional relational database terminology. A single user can be associated with multiple `channel` objects, but a single `channel` is always related to a single user. Thus this represents one-to-many relationship. `owner` attribute of a channel refers to `user` object and thus constitutes a foreign key.

If channel names should be unique only in context of a single user (e.g. `alice/news` and `bob/news` are different channels), then a composite key can be used:

```
entity channel {
  key owner: user, name;
}
```

This basically means that a pair of (`owner`, `name`) should be unique.

Finally, one might ask: what changes if we change `index owner: user` to `key owner: user`? This makes a user reference unique per `channel` table, thus there can be at most one channel per user in that case. (I.e. if `owner` is declared as a key, relationship between users and channels becomes a one-to-one relationship.)

## Operations

Now that we defined the data model, we can finally get to handling requests. As previously mentioned, Rell works with two types of requests:

1. Data-modifying requests. We call them `operations` which are applied to the database state.
2. Data-retrieving requests. We call them `queries`.

But for both types of requests we are going to need to refer to things in the database, so let's consider relational operators first.

## Creating objects

First, let's look how we create objects:

```
create user (pubkey=x
  ↪ "0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15",
  name="Alice");
```

This is essentially the same as `INSERT` operation in SQL, but the syntax is a bit different. Rell is smart enough to identify the connection between arguments and attributes based on their type. `x"..."` notation is a hexadecimal `byte_array` literal which is compatible with `pubkey` type. On the other hand, `name` is provided via `text` literal. Thus we can write:

```
create user ("Alice", x
  ↪ "0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15");
```

The order of arguments does not matter here, they are matched with attributes based on types.

## Finding objects

How do we find that object now?

```
val alice = user @ { .name=="Alice" };
```

- The `@` operator retrieves a single record (or an object in this case) satisfying the search criteria you provided. If there is no such record, or more than one exists, it raises an error. It's recommended to use this construct when an operation needs a single record to operate on. If this requirement is violated the operation will be aborted and all its effects will be rolled back. Thus it is a succinct and effective way to deal with requirements.
- `val` defines a read-only variable which can later be used in an expression. A variable defined using `var` can be reassigned later.

If you want to retrieve a list of users, you can use the `@*` operator. For example:

```
val all_users = user @* {};
```

This returns a list of all users (since no filter expression was provided, all users match it). Value declarations can include a type, for example, we can specify that `all_users` is of type `list<user>` like this:

```
val all_users: list<user> = user @* {};
```

Since the Rell compiler knows a type of every expression it does not really need a type declaration, however, if one is provided, it will check against it. Type declarations are mostly useful as documentation for programmers reading the code and should be omitted in cases where there is no ambiguity.

Both `@` and `@*` correspond to `SELECT` in SQL.

## Operation example

Let's make an operation which allows a user to create a new channel:

```
operation register_channel (user_pubkey: pubkey, channel_name: name) {  
    require( op_context.is_signer(user_pubkey) );  
    create channel (  
        owner = user@{.pubkey == user_pubkey},  
        name = channel_name  
    );  
}
```

Let's go through this line by line. First we declare the operation name and a list of parameters:

```
operation register_channel (user_pubkey: pubkey, channel_name: name) {
```

This is very similar to a function definitions in other languages. In fact, an operation is a function of a special kind: it can be invoked using a blockchain transaction by its name. When invoking `register_channel`, the caller must provide two arguments of specified types, otherwise it will fail.

```
require( op_context.is_signer(user_pubkey) );
```

We don't want Alice to be able to pull a prank on Bob by registering a channel with a silly name on his behalf. Thus we need to make sure that the transaction was signed with a key corresponding to the public key specified in the first parameter. (In other words, if Bob's public key is passed as `user_pubkey`, the transaction must also be signed by Bob, that is, Bob is a signer of this transaction.) This is a common pattern in Rell – typically you specify an actor in a parameter of an operation and in the body of the operation you verify that the actor was actually the signer. `require` fails the operation if the specified condition is not met.

`create channel`, obviously, creates a persistent object `channel`. You don't need to explicitly store it, as all created objects are persisted if operation succeeds.

`user@{.pubkey=user_pubkey}` – now we retrieve a user object by its pubkey, which should be unique. If no such user exists the operation will fail. We do not need to test for that explicitly as `@` operator will do this job.



Rell can automatically find attribute names corresponding to arguments using types. As `user` and `name` are different types, `create channel` can be written like this:

```
create channel (user@{.pubkey=user_pubkey}, channel_name);
```

## Function

Sometimes multiple operations (or queries) need the same piece functionality, e.g. some kind of a validation code, or code which retrieves objects in a particular way. In order to not repeat yourself you can use `function`. Functions work similarly to operations: they get some input and can perform validations and work with data. Additionally, they also have a return type which can be specified after the list of parameters. For example, if you want to allow the user of a channel to change the name of the channel itself:

```
// We added mutable specifier to channel's attribute "name" to make name editable.
// Note that in case both an attribute and a key need to be declared.

entity channel {
    mutable name;
    key name;
    index owner: user;
}

function get_channel_owned_by_user(user_pub: pubkey, channel_name: name): channel {
    val user = user@{.pubkey == user_pub};
    return channel@{channel_name, .owner == user};
}

operation change_channel_name(signer: pubkey, old_channel_name: name, new_channel_
↪name: name) {
    require(op_context.is_signer(signer));
    val channel_to_change = get_channel_owned_by_user(signer, old_channel_name);
    update channel@{channel == channel_to_change}(.name = new_channel_name);
}
```

In the function `get_channel_owned_by_user` the code first retrieves a user with given public key and returns a channel owned by the retrieved user with the given channel name. Operator `@` expects exactly one object to be found (see [Cardinality](#) for more information.), thus you can be sure that in case there is no user or channel with such a pubkey or name the function will fail and so will the operation that is calling it. Finally, the function returns the channel instance that was validated, saving the developer the hassle to check owner every time a channel is retrieved.

Please note that you must mark the attribute `name` with the keyword `mutable`. This is because only the fields which are declared mutable can be changed using the update statement.

## Query

Storing data without the ability to access it again would be useless. Let's consider a simple example - retrieving channel names for a user with a certain name:

```
query get_channel_names (user_name: name) {
    return channel @* {
        .owner == user@{.name==user_name}
    } (.name);
}
```

Here you see a selection operator you're already familiar with – `@*`. We select all the channels with a given owner (which we first find by name).

Then we extract name attribute from retrieved objects using the `(.name)` construct.

Note that since we only need `name` from `channel`, is also possible to write

```
query get_channel_names (user_name: name) {
  return channel @* {
    .owner == user@{.name==user_name}
  }.name;
}
```

## Relational expressions

In general, a relational expression consists of five parts, some of which can be omitted:

FROM OPERATOR { WHERE } (WHAT) LIMIT

1. *FROM* describes where data is taken from. It can be a single entity, such as just `user`. Or, it can be combination of multiple entities, e.g. `(user, channel)`. In the latter case, conceptually we are dealing with a Cartesian product, which is a set of all possible combinations. But, typically *WHERE* part will then provide a condition which defines a correspondence between objects of different entities. E.g. one can select such `(user, channel)` combinations where `user` is an owner of the `channel`. This works same way as `JOIN` in `SQL`, in fact, the optimizer will typically translate it to `JOINS`.
2. *OPERATOR* – there are different operators depending on required cardinality. They are:
  - `@` – exactly one, returns a value
  - `@*` – any number, returns a list of values
  - `@+` – at least one, returns a list of values
  - `@?` – one or zero, returns a nullable value
3. *WHERE* describes how to filter the *FROM* set. So, you would use your search criteria as well as `JOINS`.
4. *WHAT* describes how to process the set, for doing a projection, aggregation or sorting. If it is omitted then members of the set are returned as they are.
5. *LIMIT* for operators which return a list, limits the number of elements returned.

In `SQL`, the logical processing order does not match the order in which clauses are written, for example, `FROM` is logically processed before `SELECT` even though `SELECT` comes first. (`SQL` logical processing order can be found e.g. in [SQL Server documentation](#)).

The order of components of a relational expression in `Rel` matches the logical processing order. So, first a set is defined, then it is filtered, and then it is post-processed. Of course, the query planner is allowed to perform operations in a different order, but that shouldn't affect the results. Thus a relational expression can be understood as a kind of a pipeline.

Let's see some examples of relational expressions. Suppose in addition to `user` and `channel` entities we provided before, we also have:

```
entity message {
  index channel;
  index timestamp;
  text;
}
```

We can retrieve all messages of a given user:

```
(channel, message) @* {
    channel.owner == given_user, message.channel == channel
} (message.text);
```

So, basically, we join `channel` with `message`. We can shorten the expression using entity aliases:

```
(c: channel, m: message) @* { c.owner == given_user, m.channel == c } (m.text, m.
↳timestamp)
```

We can easily read this expression left to right:

- consider all pairs  $(c, m)$  where  $c$  is channel and  $m$  is message
- find those where  $c.owner$  equals `given_user` and  $m.channel$  equals  $c$
- extract `text` and `timestamp` from  $m$

The result of this expression is a list of tuples with `text` and `timestamp` attributes.

The above expression can be easily modified to retrieve the latest 25 messages:

```
(c: channel, m: message) @* {
    c.owner == given_user, m.channel == c
} (m.text, @sort_desc m.timestamp) limit 25
```

Here we sorted results by timestamp in a descending order using `@sort_desc` and limited the number of returned rows.

## Composite indices

We can also select recent messages by adding, for example, `m.timestamp >= given_timestamp` condition to *WHERE* part. But a database cannot filter messages efficiently (that is, without considering every message) using two criteria at once unless we create a *composite index*, changing the message entity definition in the following way:

```
entity message {
    index channel, timestamp;
    text;
}
```

Instead two separate indexes we got one composite index. The idea here is that we want to retrieve not the latest messages overall, but the latest messages *for a given channel*. Thus, we need to order messages by channels first. Paged retrieval can be done using the following query:

```
query get_next_messages (user_name: name, upto_timestamp: timestamp) {
    val given_user = user@{user_name};
    return (c: channel, m: message) @* {
        c.owner == given_user, m.channel == c, m.timestamp < upto_timestamp
    } (m.text, -sort m.timestamp) limit 25;
}
```

This can be used in an app like Twitter. A visitor might first retrieve the latest 25 messages, then go further – in which case the client will send a query with a timestamp of the oldest message retrieved.

To understand why this can work efficiently, consider that the index stores an ordered collection of pairs. For example:

```
1. (channel_1, 1000000) -> m1
2. (channel_1, 1000050) -> m3
3. (channel_1, 1000100) -> m5
4. (channel_2, 1000025) -> m2
5. (channel_2, 1000075) -> m4
```

A database can efficiently find a place which corresponds to a given timestamp in a given channel and traverse the index through it.

---

**Note:** It's worth noting that all SQL databases work this way, this feature is not unique to Rel. But in a decentralized system resources are typically precious, thus it is important for Rel programmers to understand the query behavior and use indices efficiently.

---

## 2.2.2 Client Side

Ok, so now we have a basic understanding of how Rel works, but users don't typically query Rel directly. They instead perform actions in a frontend that sends transactions to the Rel backend with the help of postchain client code. Because state gets stored on a blockchain, the postchain client needs to sign transactions and perform other blockchain-related operations before it sends transactions to Rel.

This client tutorial is a continuation on the quickstart "city" example. In this section we illustrate how to send transactions to and retrieve information from a blockchain node running Rel.

### Try the example code

First of all, we need to add a query to Rel source file:

```
query is_city_registered(city_name: text): boolean {
  return (city @? { city_name }) != null;
}
```

Clicking 'Start node' will start a Postchain node in a single-node mode which is convenient for testing.

The node builds blocks when there are transactions, or at least once every 30 seconds. It also has REST API we can interact with to submit transactions and retrieve information.

The client code is written in JavaScript, this example uses the NodeJS environment. [postchain-client-example](#) can be downloaded using git:

```
git clone https://bitbucket.org/chromawallet/postchain-client-example.git
```

To run it, execute:

```
npm install
node index.js
```

This will create a transaction, sign it and submit it to a node. Once the transaction is added to a block, the client will perform a query.

Now let's see how this client code can be implemented:

## Install the client

**Note:** It is recommended that you write your javascript client outside of the Web IDE, as it is primarily designed to set up a Rell backend. Open up a text editor or an IDE and follow the steps.

We assume you have `nodejs` installed. The client library is called `postchain-client` and can be installed from npm.

Create a new directory for your test. Open a terminal in the new directory, initialize npm and install the client.

```
npm init -y
npm install postchain-client --save
```

## Connect to the node

To connect to a Postchain node we need to know its REST API URL and blockchain identifier. DevPreview bundle comes with following defaults:

```
const pcl = require('postchain-client');

// using default postchain node REST API port
// On rellide-staging.chromia.dev, check node log for api url
const node_api_url = "http://localhost:7740";

// default blockchain identifier used for testing
const blockchainRID =
  ↪ "78967baa4768cbcef11c508326ffb13a956689fcb6dc3ba17f4b895cbb1577a3";

const rest = pcl.restClient.createRestClient(node_api_url, blockchainRID, 5);
```

**Note:** The blockchainRID is unique for each setup. For port 7740, it can be found at [http://localhost:7740/brid/iid\\_1](http://localhost:7740/brid/iid_1)

Once we set up the information about the the REST Client connection, we can create the gtxClient connection. This in particular, needs to receive the previous REST connection, the blockchainRID in Buffer format and an array the names of the operations that you want to call (at the moment this can be left empty):

```
const gtx = pcl.gtxClient.createClient(
  rest,
  Buffer.from(blockchainRID, 'hex'),
  []
);
```

Now that the connection is set, you can start to create transactions and queries.

## Make a transaction (with operations inside)

You need to create the transaction client side, sign it with one or more keypairs, send it to the node and wait for it to be included into a block.

First, let's create the transaction and specify the public key of the person(s) that will sign it. To create a random user keypair on the go you can use `makeKeyPair()` function.

```
const user = pcl.util.makeKeyPair();
const tx = gtx.newTransaction([user.pubKey]);
```

Once the transaction has been created it is possible to call as many operations as you want.

```
tx.addOperation('insert_city', "Toulouse");
tx.addOperation('insert_city', "Stockholm");
tx.addOperation('insert_city', "Bologna");
/* etc */
```

Now, all is left is to sign and post the transaction

```
tx.sign(user.privKey, user.pubKey);
tx.postAndWaitConfirmation();
```

Note: `tx.postAndWaitConfirmation()` returns a promise, and thus can be `await`-ed.

### Query

Queries also make use of `gtx` client.

`gtx.query` accepts as first parameter the name of the query as specified in the module and then an object with as parameter name the variable name as specified in the query module.

E.g:

```
function is_city_registered(city_name) {
    return gtx.query("is_city_registered", {city_name: city_name});
}
```

will work with query specified in the Rel file:

```
query is_city_registered(city_name: text): boolean {
    return (city @? { city_name }) != null;
}
```

Note: `gtx.query(queryName, queryObject)` also returns a promise.

### Examples and further exercises

For now we have covered the basics of working with Rel. In the next section, we will go over the Rel plugin in Eclipse and how to use it. But if you want to toy around with examples in the Web IDE, you can skip ahead to the Examples chapter.

## 2.3 Eclipse IDE

Rel has a plugin for Eclipse which can be used to write Rel code and starting up a testnode with minimal hassle. In this chapter we will be introduced to this plugin, how to write a Rel program with it and how to launch a testnode with it.

## 2.3.1 Installation

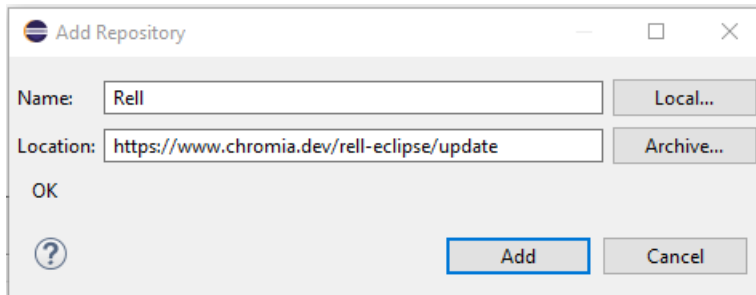
### Prerequisites

- Java 11+.
- To be able to run a Postchain node: PostgreSQL 10 or later.
- Eclipse IDE (Eclipse can be downloaded [here](#).)

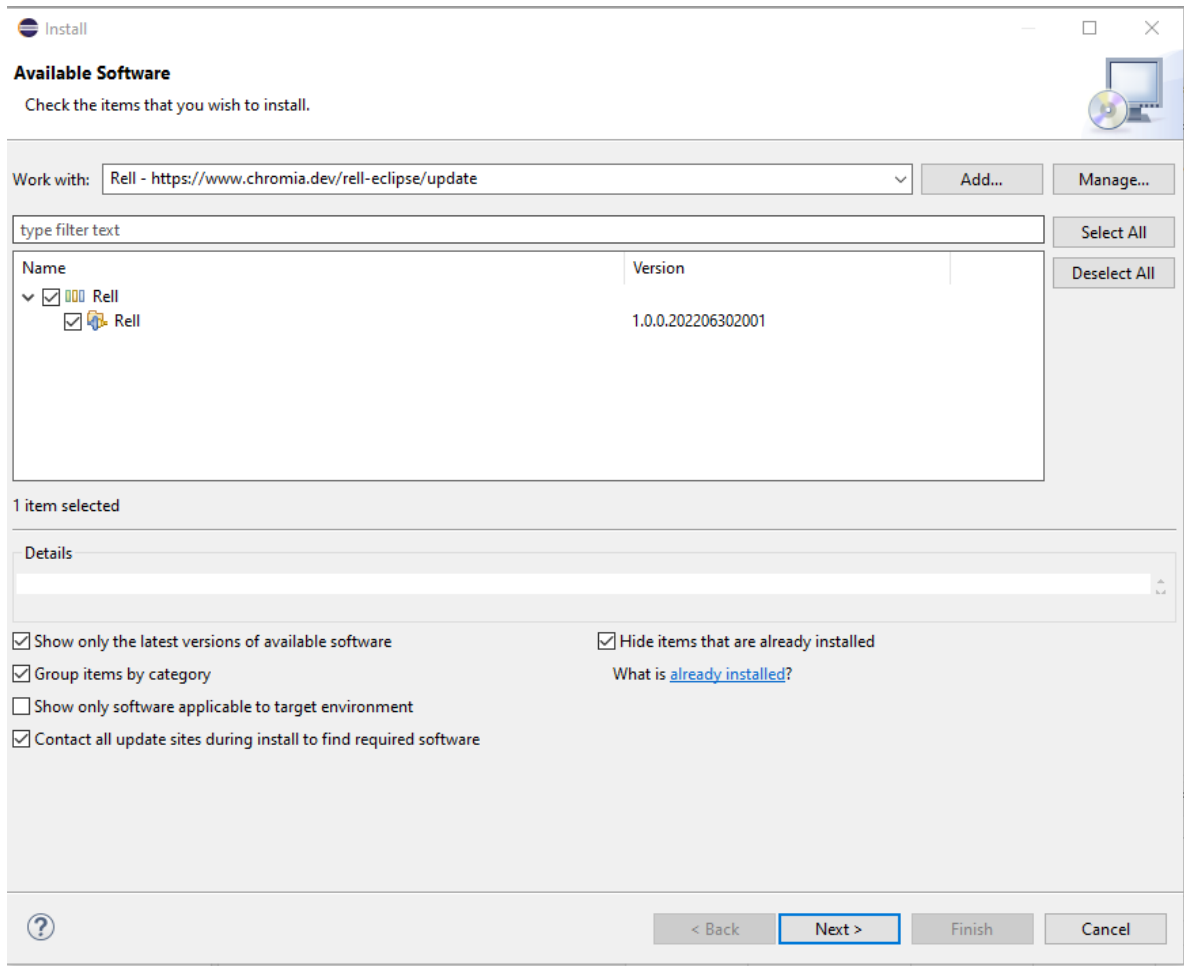
### Adding the Rell plugin to the Eclipse IDE

Once you have downloaded the Eclipse IDE, the next step is to add the Rell plugin.

1. Go to the menu **Help - Install New Software...**
2. In the Install dialog, click **Add...**, then type:
  - Name: Rell
  - Location: <https://www.chromia.dev/rell-eclipse/update>

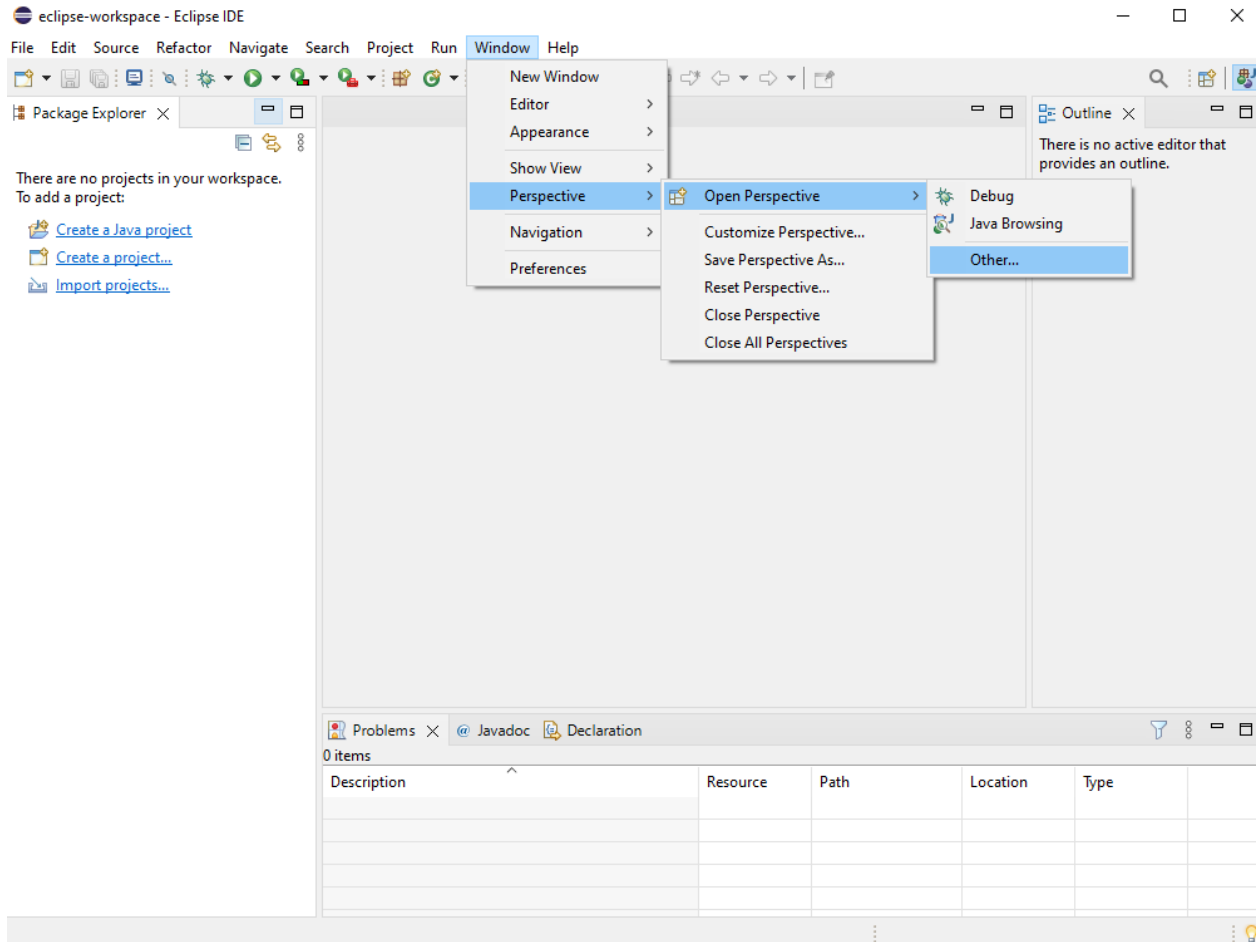


1. Rell shall appear in the list. Select it and click **Next >**, then **Finish**.



2. If a warning “You are installing software that contains unsigned content” appears, click **Install anyway**.
3. Click **Restart Now** when asked to restart Eclipse IDE.
4. Switch to the Rel perspective. Menu **Window - Perspective - Open Perspective - Other...**, choose **Rel**.





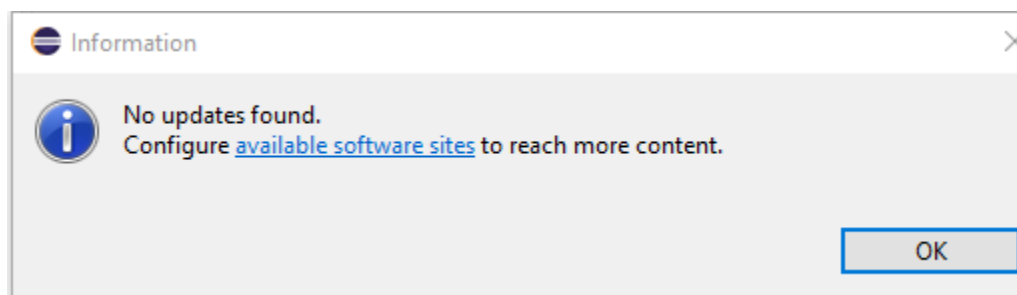
## Enabling automatic updates for the plugin

When a new version of the Rell plugin is released, it has to be updated in Eclipse. If Rell update URL has already been configured, Eclipse will check for updates automatically once in a while, and show a message when a plugin can be updated.

To manually check for updates:

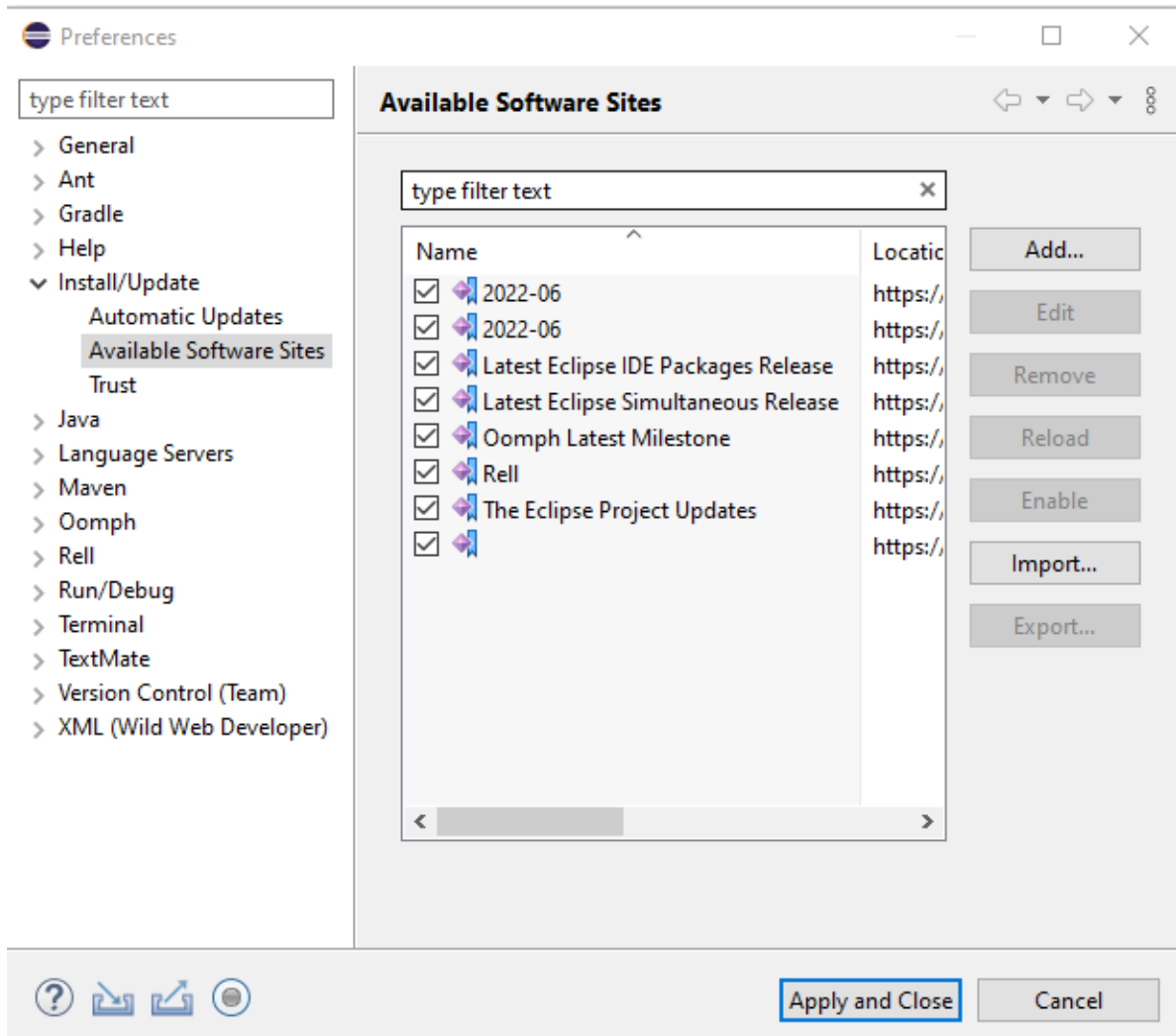
1. Menu: **Help - Check for Updates**.
2. If it shows that a new version of Rell plugin is available, install it.

If “No updates found” message is shown, check that Rell update site is set up.



1. Click **available software sites** link in the message dialog.

2. If there is no RelI in the list, click **Add...** to add it.



3. Specify:

- Name: RelI
- Location: <https://www.chromia.dev/rell-eclipse/update>

and click **Add**.

If RelI update site is in the list, but “No updates found” message is shown, try to reload the site:

1. Click **available software sites** link in the message dialog.
2. Click **Reload**.

If still no updates are shown, your RelI plugin must be already up-to-date.

## 2.3.2 Database Setup

## Database Setup

Rell requires PostgreSQL 10 to be installed and set up. The IDE can work without it, but will not be able to run a node. A console app or a remote postchain app can be run without a database, though.

Default database configuration for Rell is:

- database: postchain
- user: postchain
- password: postchain

## Ubuntu (Debian)

Install PostgreSQL:

```
sudo apt-get install postgresql
```

Prepare a Postgres database:

```
sudo -u postgres psql -c "CREATE DATABASE postchain;" -c "CREATE ROLE postchain LOGIN_
↳ ENCRYPTED PASSWORD 'postchain'; GRANT ALL ON DATABASE postchain TO postchain;"
```

## MacOS

Install PostgreSQL:

```
brew install postgresql
brew services start postgresql
createuser -s postgres
```

Prepare a Postgres database:

```
psql -U postgres -c "CREATE DATABASE postchain;" -c "CREATE ROLE postchain LOGIN_
↳ ENCRYPTED PASSWORD 'postchain'; GRANT ALL ON DATABASE postchain TO postchain;"
```

**Note:** If you get an error saying that peer authentication failed, you will have to change authentication method from peer to md5. this can be done inside the pg\_hba.conf file of your psql database.

## Docker

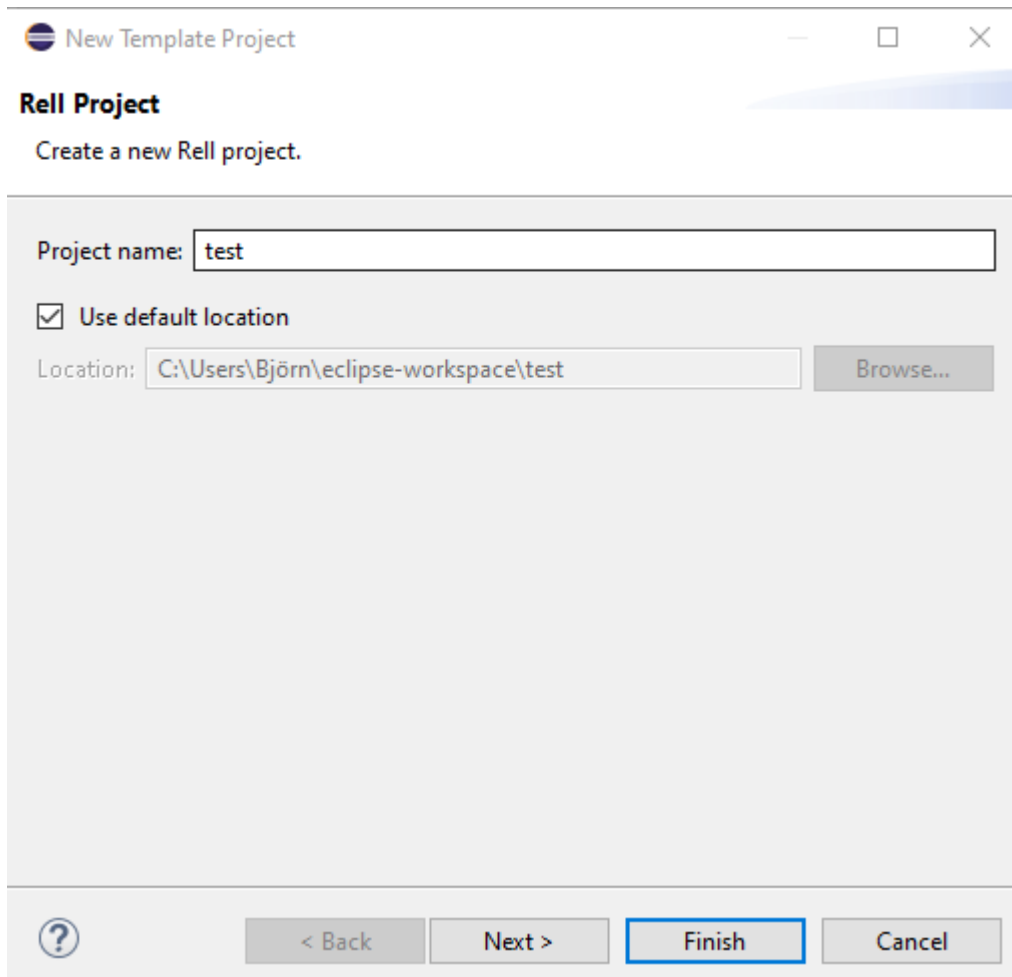
```
docker run --name postchain -e POSTGRES_USER=postchain -e POSTGRES_PASSWORD=postchain_
↳ -p 5432:5432 -d postgres
```

## 2.3.3 Using Rell in Eclipse

Now that we have everything installed, we should try to make a simple hello world in Rell.

## Hello World in Rel

1. Switch to the Rel perspective, if not done already. Menu: **Window - Perspective - Open Perspective - Other...**, choose **Rel**.
2. Create a project:
  - Menu **File - New - Rel Project**.
  - Enter a project name `test` and click **Finish**.



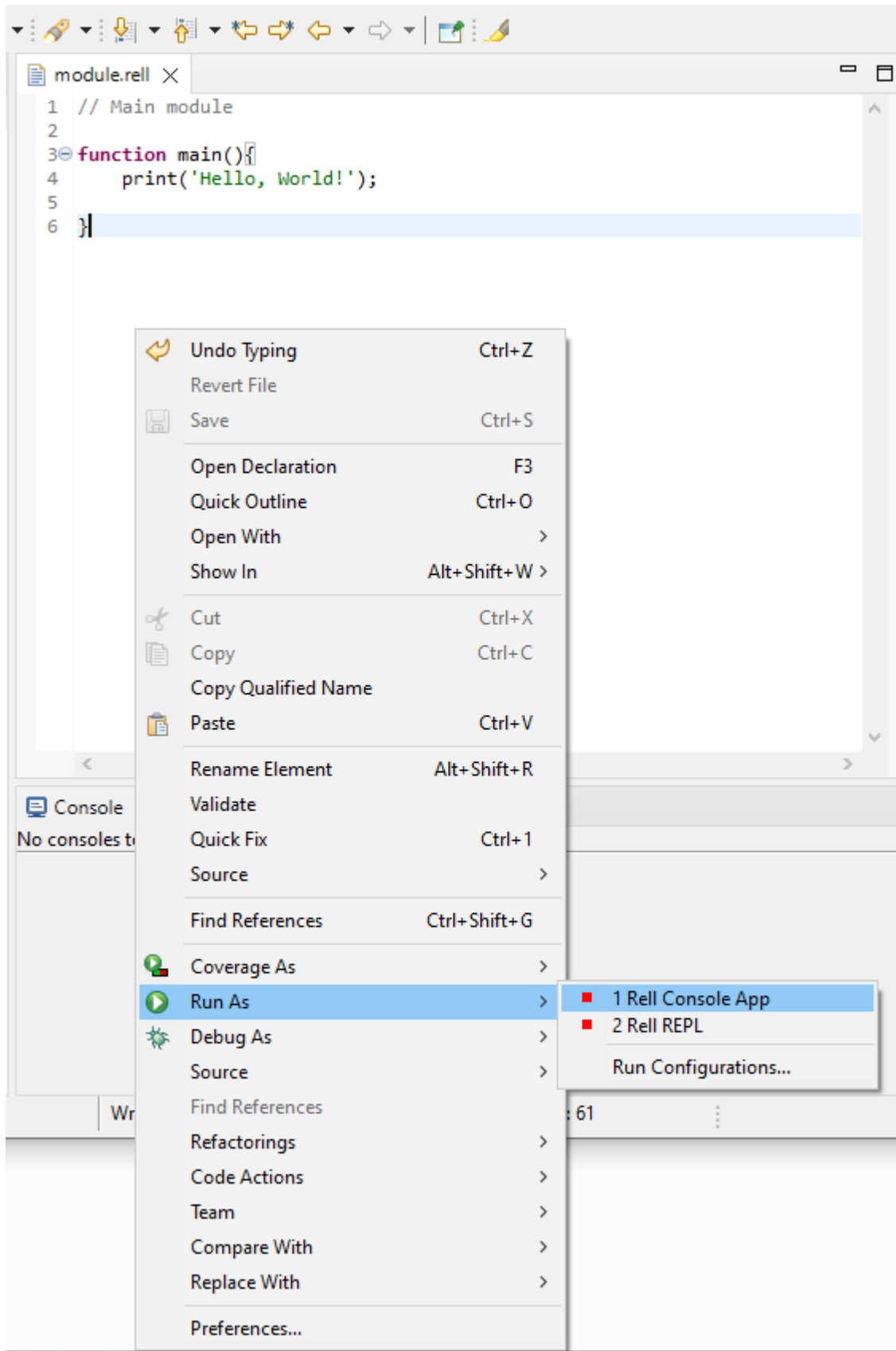
A Rel project with a default directory structure shall be created.

3. Open the `module.rell` file:
  - click on the project folder `test` folder and then `src-test`.
  - Click on the `module.rell` file to open the editor
4. Write following code in the editor:

```
function main() {  
    print('Hello, World!');  
}
```

Save: **File - Save** or CTRL-S (S).

5. Run the program: right-click on the editor and choose **Run As - Reli Console App**.

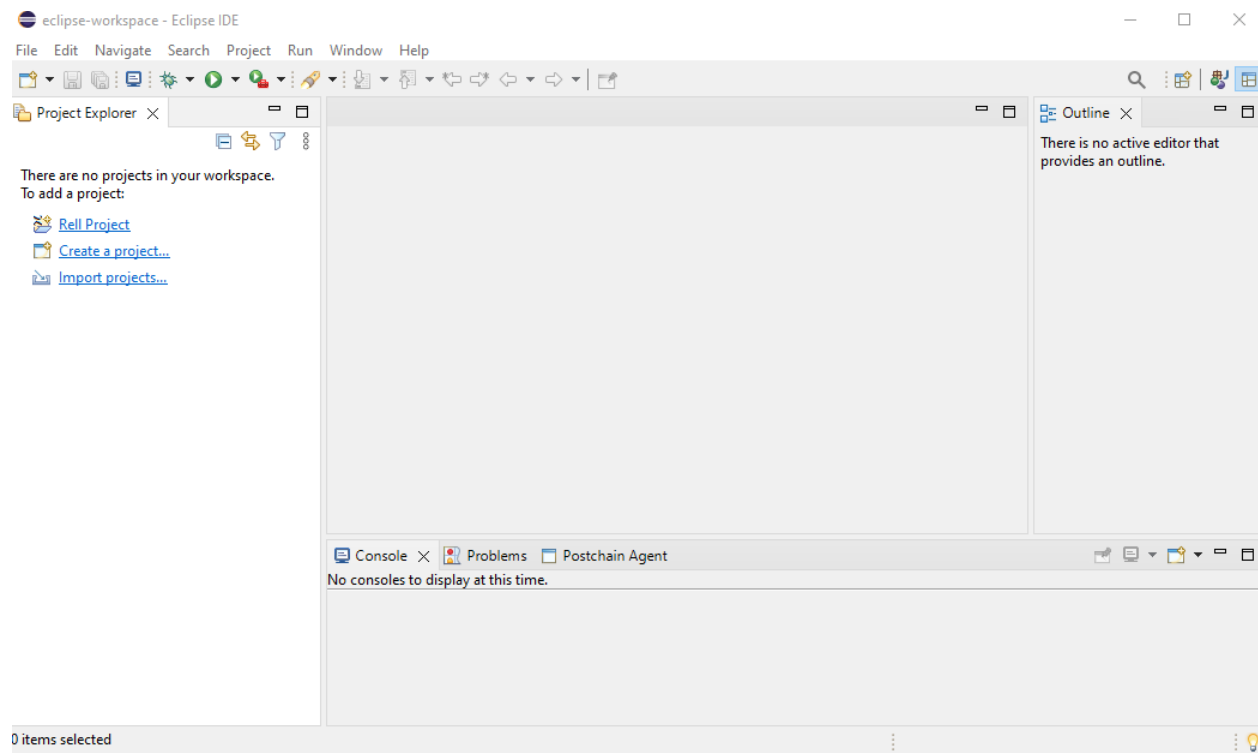


6. The output “Hello, World!” should be shown in the Console view.

## IDE Overview

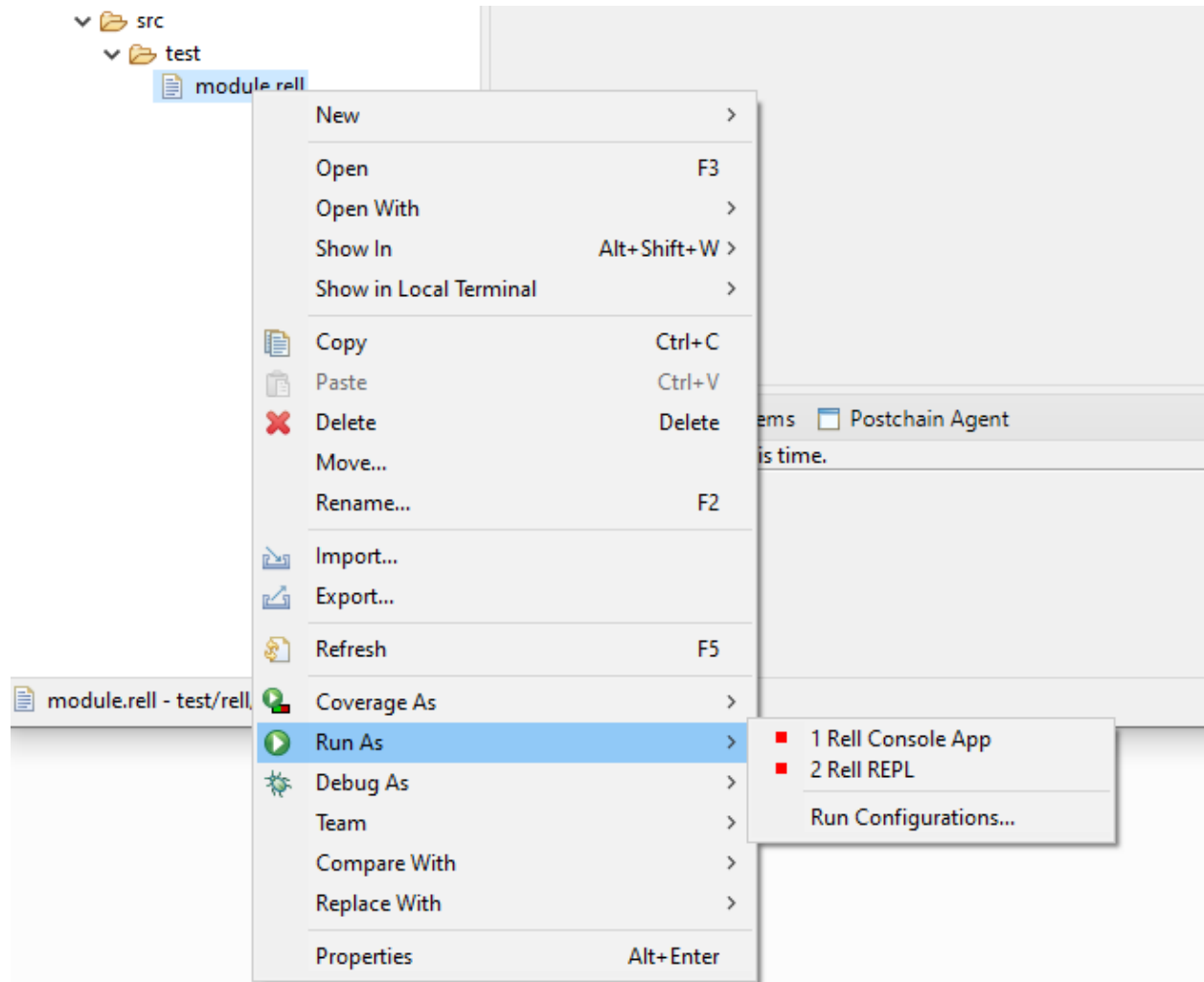
Eclipse IDE window consists of different views. Every view has its own tab. By default, Rell IDE has following views:

- **Project Explorer** - shows projects and their directory trees.
- **Problems** - shows compilation warnings and errors.
- **Console** - console output (when running programs).
- **Outline** - shows the structure of a selected Rell file.



## Running Applications

Right-click on a file (or an editor) and choose **Run As**. Run options available for the file will be shown.



Alternatively, use keyboard shortcut CTRL-F11 (F11).

## Rel Console App

Executes a `*.rell` file as a stand-alone console program, not as a module in a Postchain node.

The program must contain a function (or operation, or query) called `main`, which will be the entry point. The output is displayed in the Console view.

The name of the main function and its arguments can be specified in a *run configuration*.

## Database connection

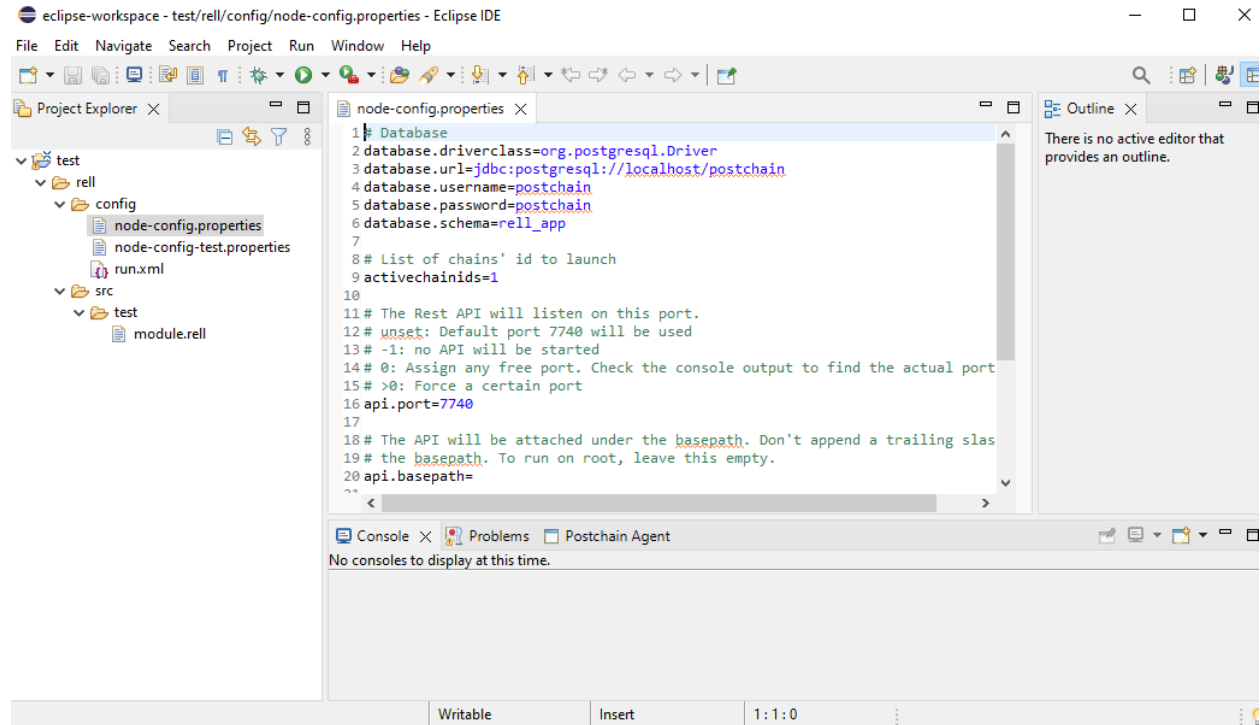
By default, the program is executed without a database connection, and an attempt to perform a database operation will result in a run-time error.

To run a console app with a database connection, there must be a file called `console-db.properties`, `db.properties` or `node-config.properties` in the directory of the Rel file or in the `rell/config` directory of the project. The file shall contain database connection settings. For example:



```

database.driverclass=org.postgresql.Driver
database.url=jdbc:postgresql://localhost/postchain
database.username=postchain
database.password=postchain
database.schema=reII_app
    
```



When running a console app with a database connection, tables for defined classes and objects are created on start-up. If a table already exists, missing columns are added, if necessary.

How to prepare a database is described in the [Database Setup](#) section.

## ReII Postchain App

Starts a Postchain node with a configuration written in the [Run.XML](#) format. To use this option, right-click on a \*.xml file, not on a \*.reII file.

Using run.xml gives you the option to run multiple blockchains in one Postchain node.

Example of a minimal run.xml:

```

<run>
  <nodes>
    <config src="node-config.properties" add-signers="true" />
    <test-config src="node-config-test.properties"/>
  </nodes>
  <chains>
    <chain name="test" iid="1">
      <config height="0">
        <app module="test">
          </app>
        </config>
      </chain>
    </chains>
  </run>
    
```

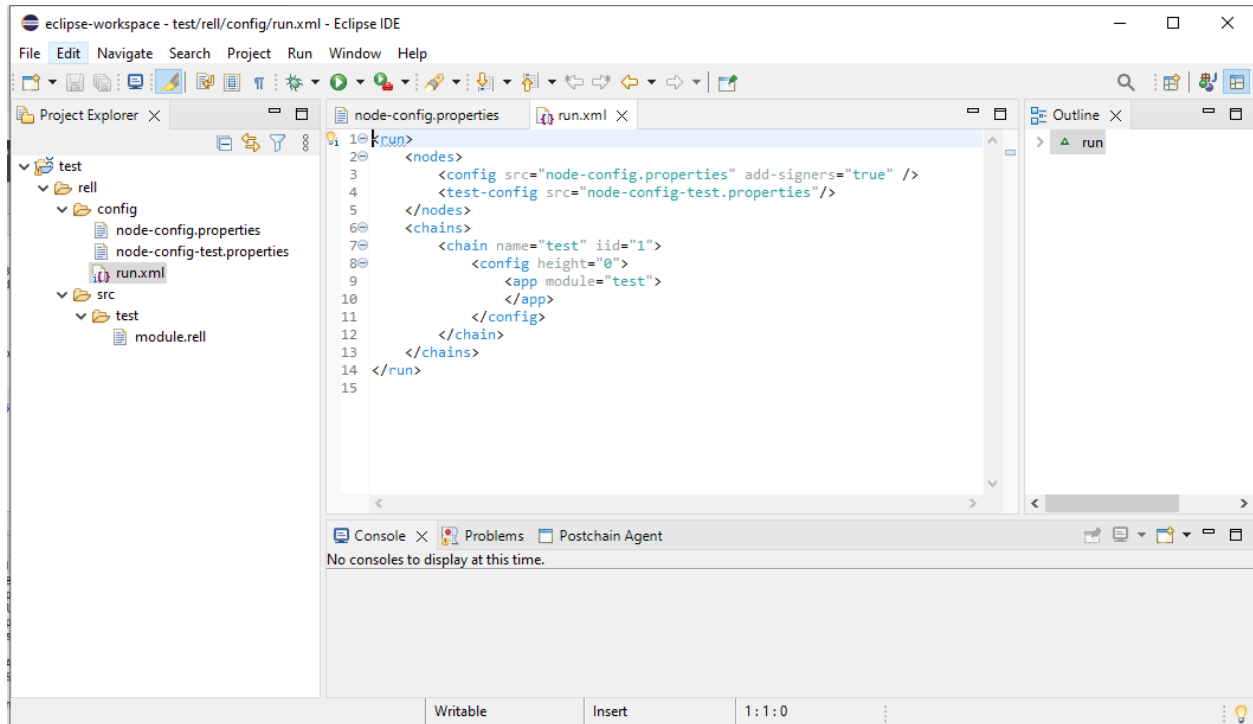
(continues on next page)

(continued from previous page)

```

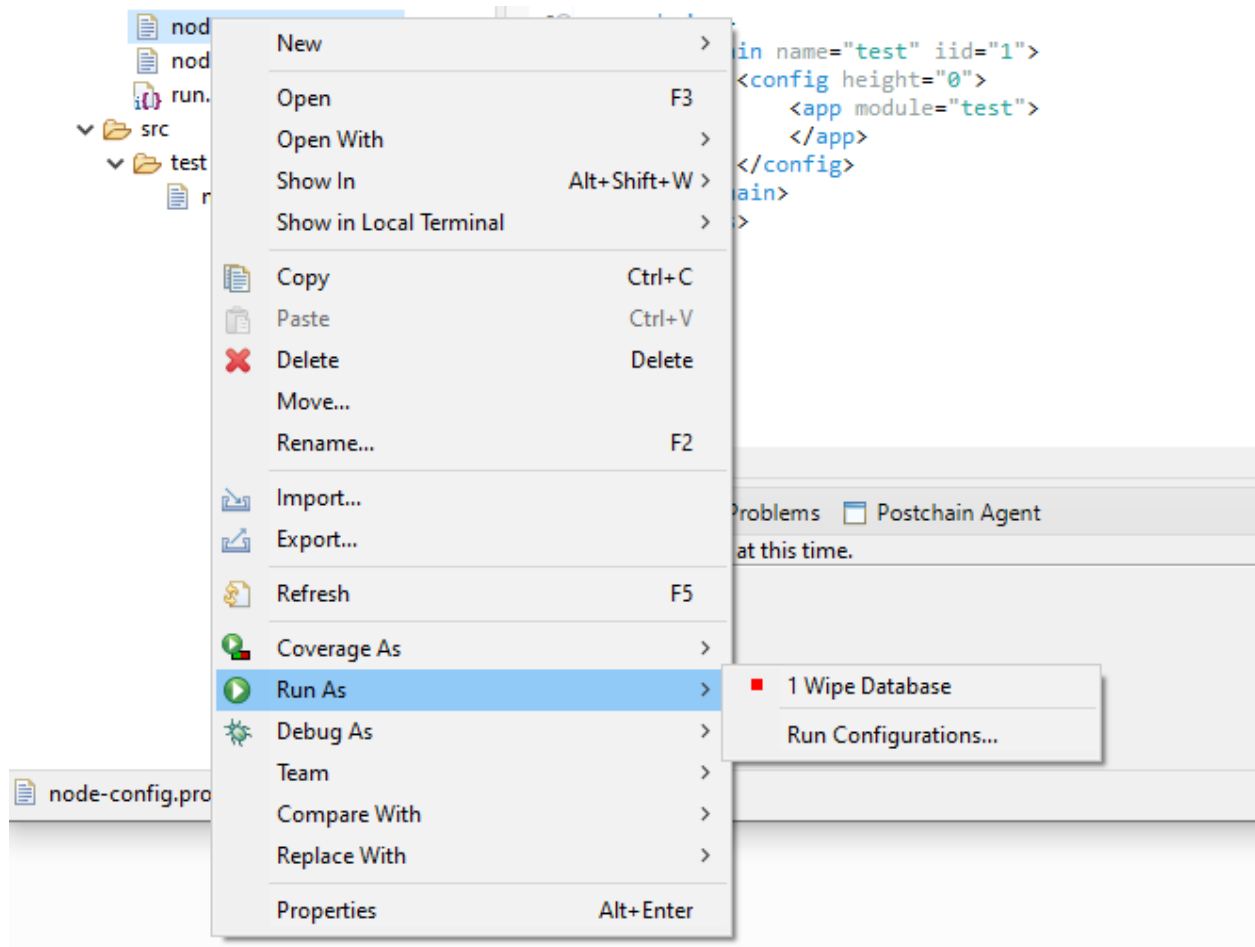
</chain>
</chains>
</run>

```



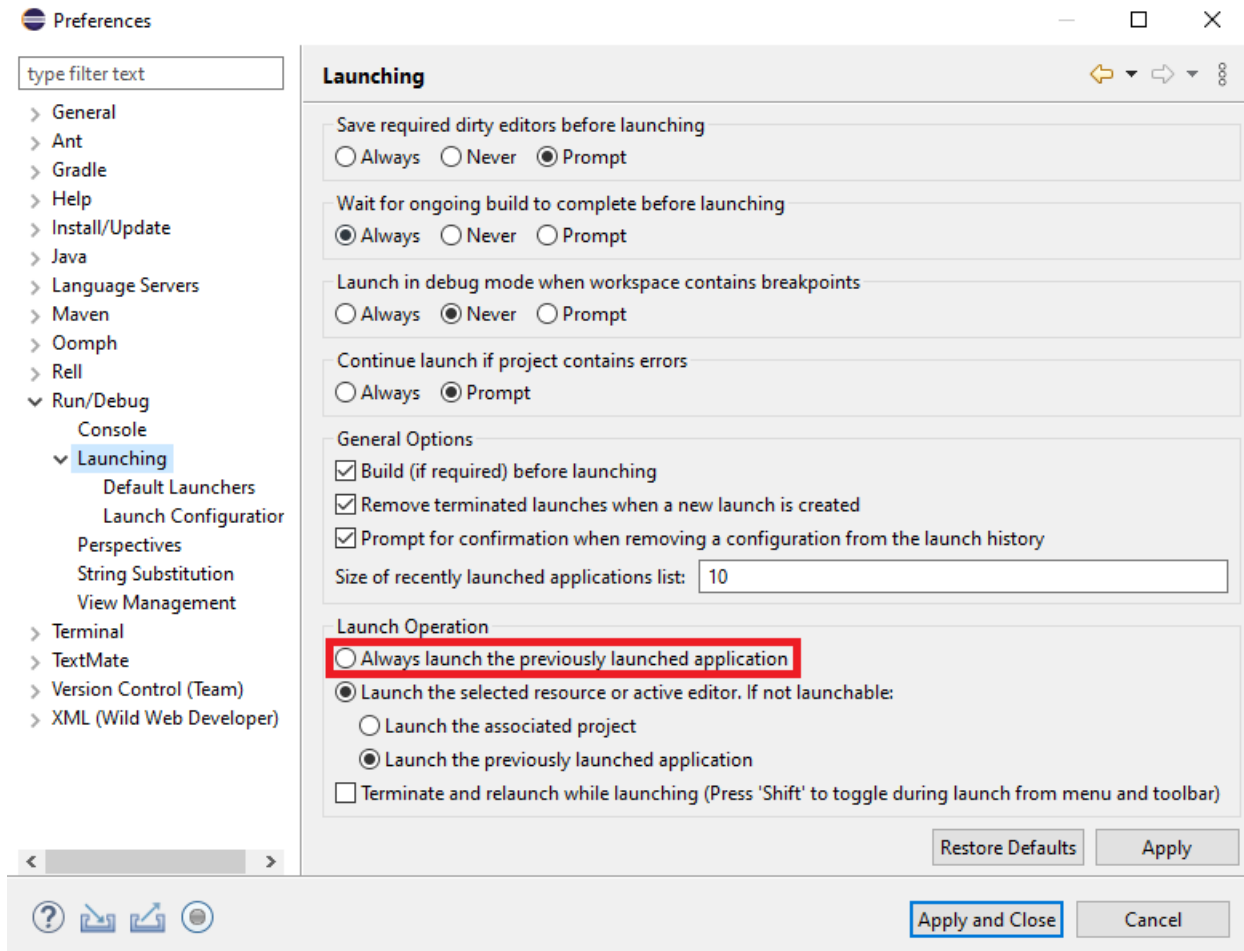
## Wipe Database

This option is available for database properties files, \*.properties (for instance, node-config.properties). Drops all tables (and stored procedures) in the database.



## Run with a Keyboard

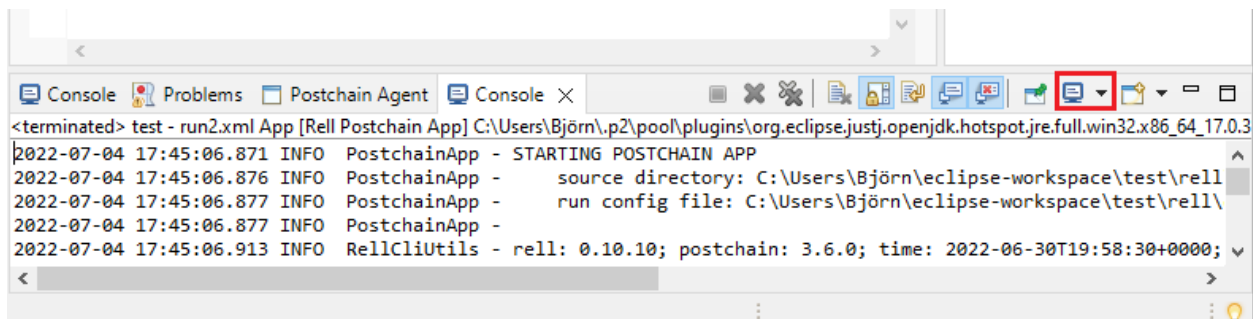
By default, CTRL-F11 (F11) shortcut runs the file of the active editor. It can be configured to run the last launched application instead, which may be more convenient, as there is no need to choose an application type. Go to the menu **Window - Preferences** (macOS: **Eclipse - Preferences**), then **Run/Debug - Launching**. In the **Launch Operation** box, choose **Always launch the previously launched application**.



## Running Multiple Apps

It is possible to run multiple applications (e. g. multiple nodes) simultaneously. For example, one can define two [Run.XML](#) configuration files that use the same Rel module, but different ports and database schemas.

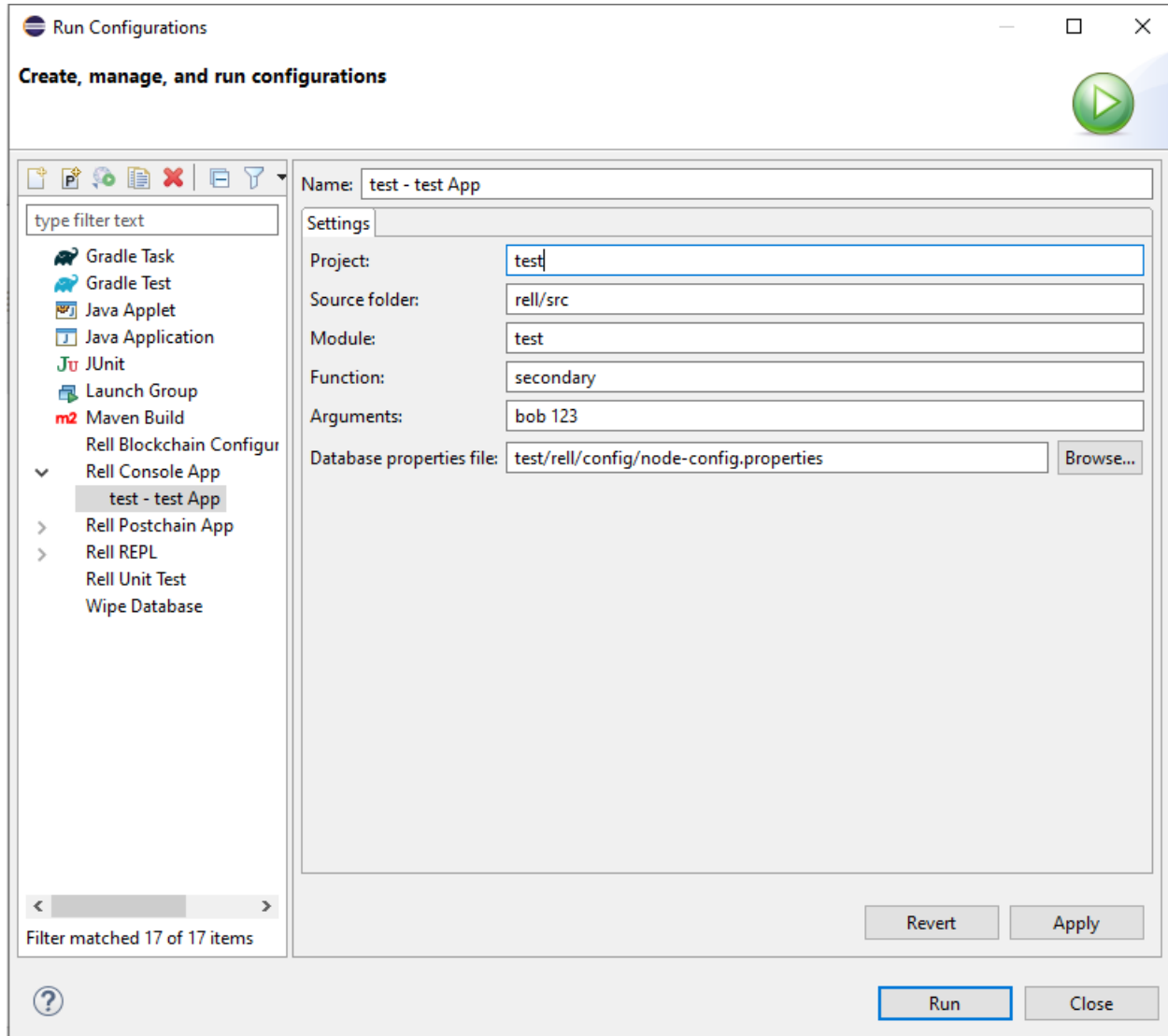
The output of all running applications will be shown in the Console view, but on different pages. It is possible to switch between the consoles of different applications using a button or a dropdown list.



## Run Configurations

To run an application, Eclipse IDE needs a run configuration, which contains different properties, like the name of the main function, arguments or blockchain RID. When running an application via the **Run As** context menu, the IDE automatically creates a run configuration with default settings if it does not exist.

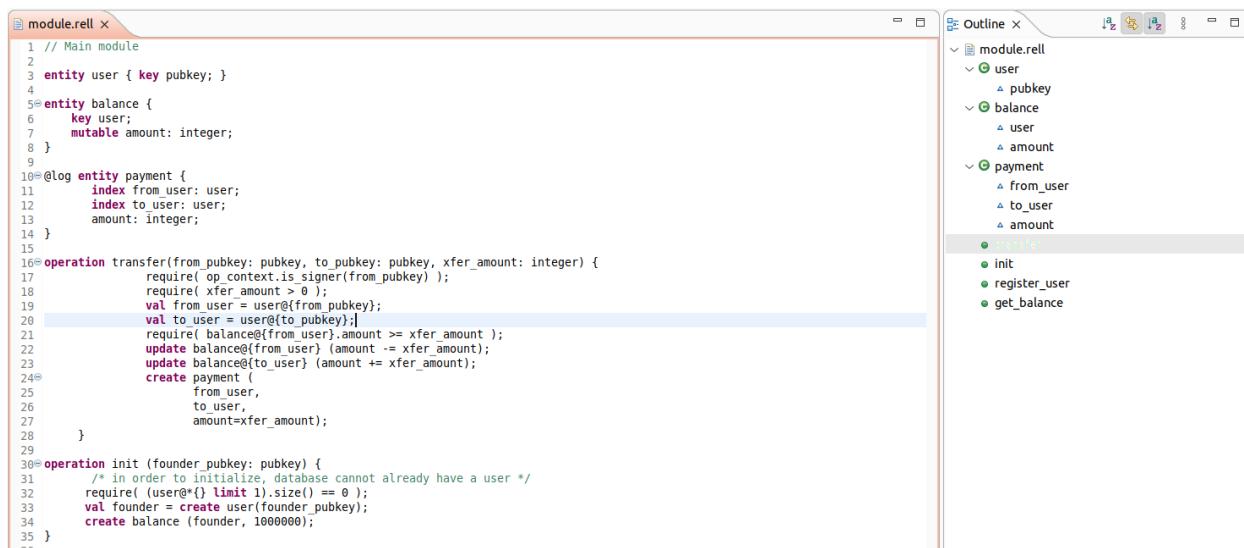
To change a run configuration, go to the menu **Run - Run Configurations...** The last launched application will be selected. Change the settings and click either **Apply** or **Run** to save the changes.



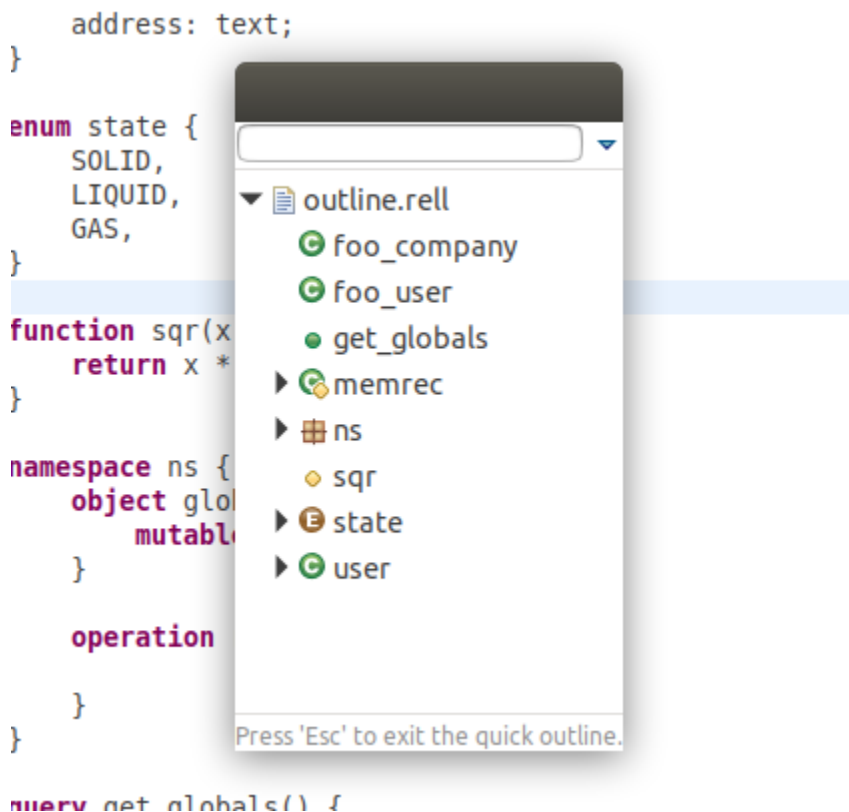
## 2.3.4 Features of the IDE

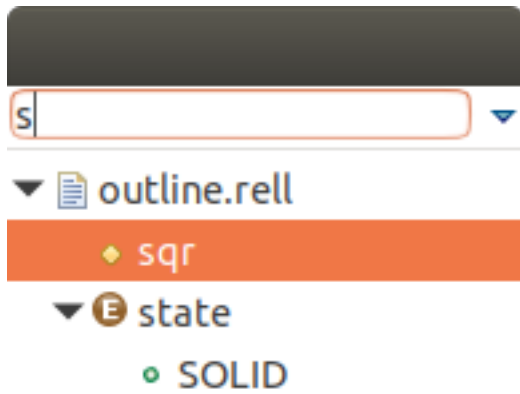
### Outline

When a Rell editor is open, the structure of its file (tree of definitions) is shown in the Outline view (which is by default on the right side of the IDE).



There is also a Quick outline window, which is activated by the CTRL-O (O) shortcut. Type a name to find its definition in the file.

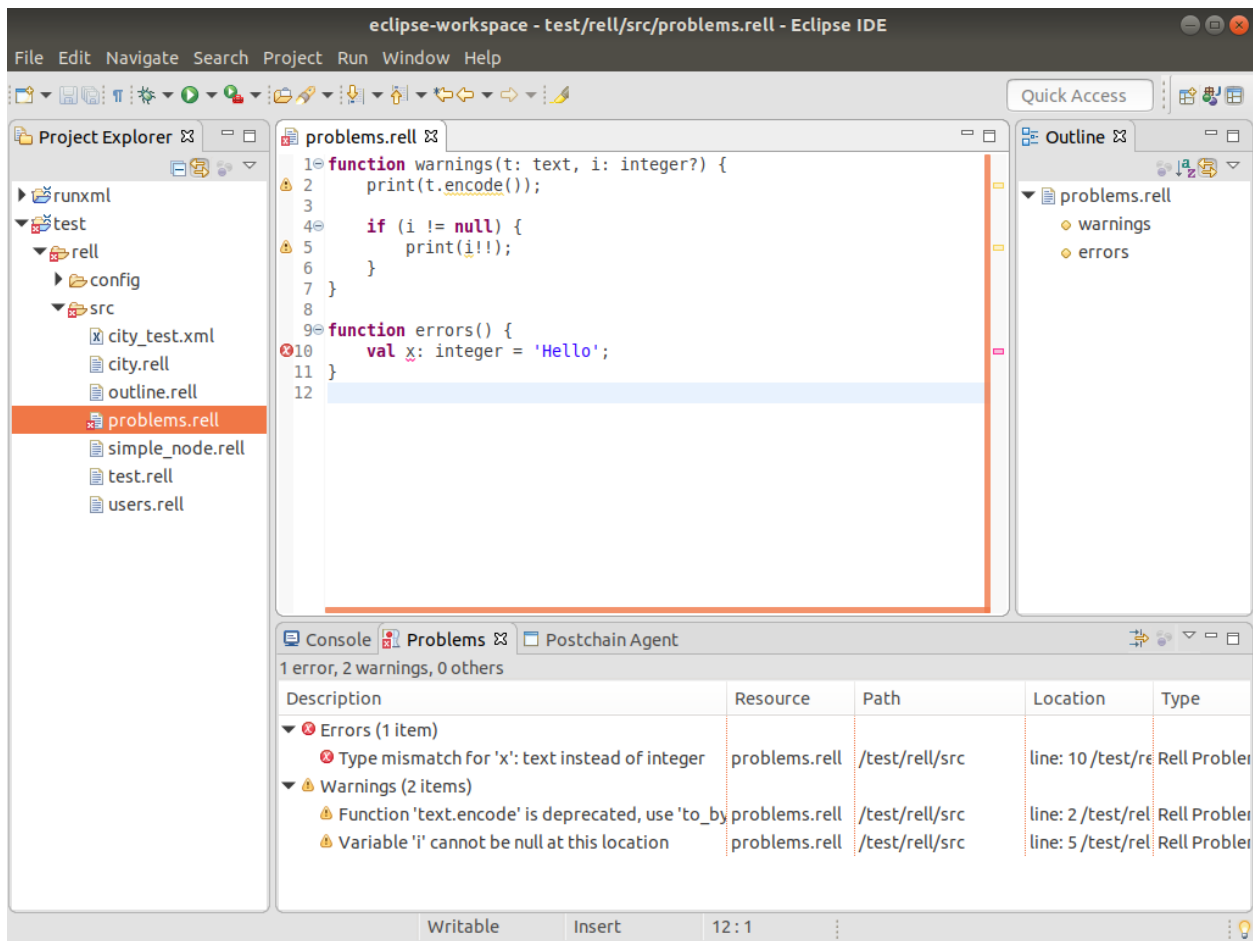




Press 'Esc' to exit the quick outline.

### Problems View

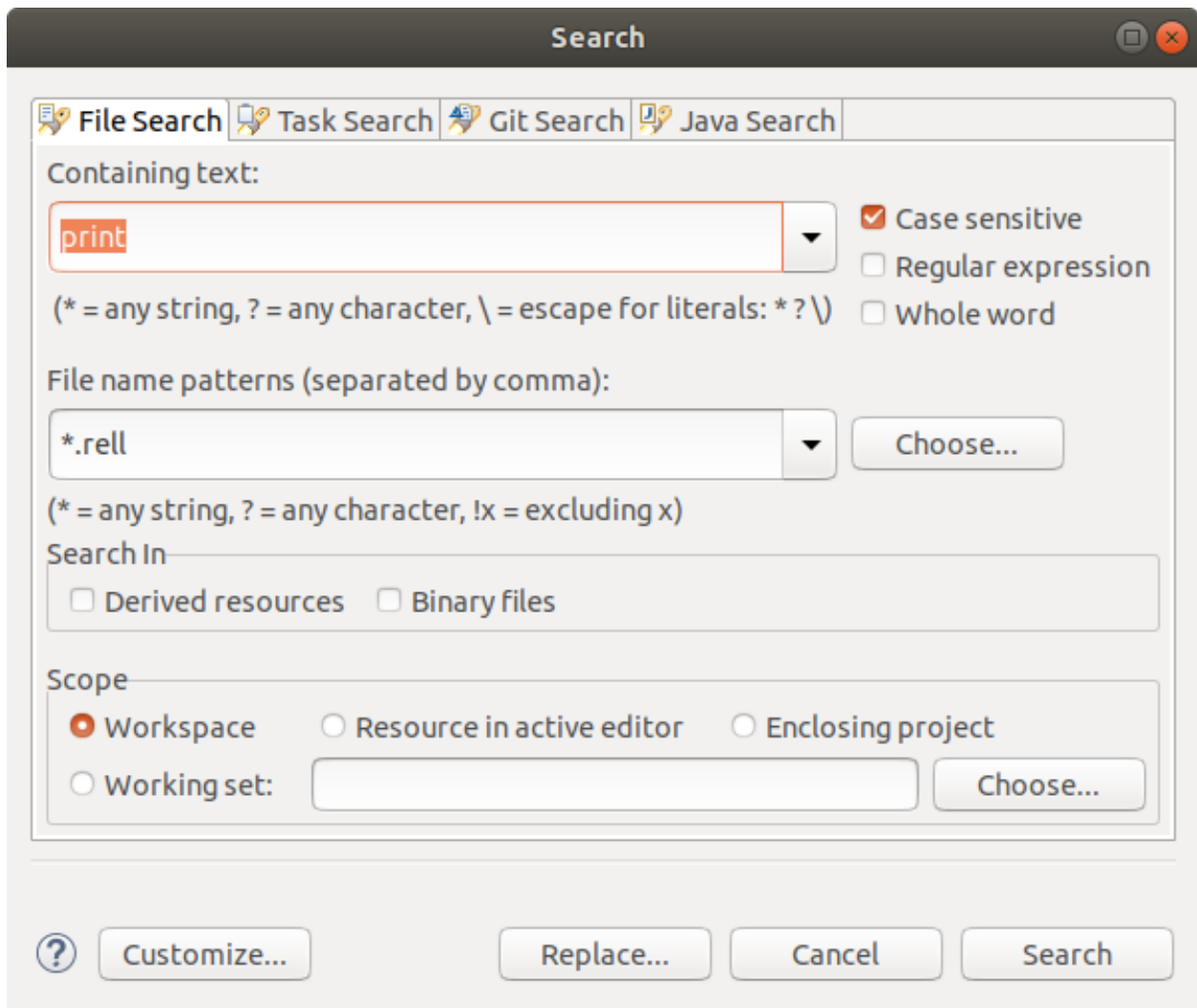
Problems view shows compiler warnings and errors found in all projects in the IDE. The list is updated when saving a file.



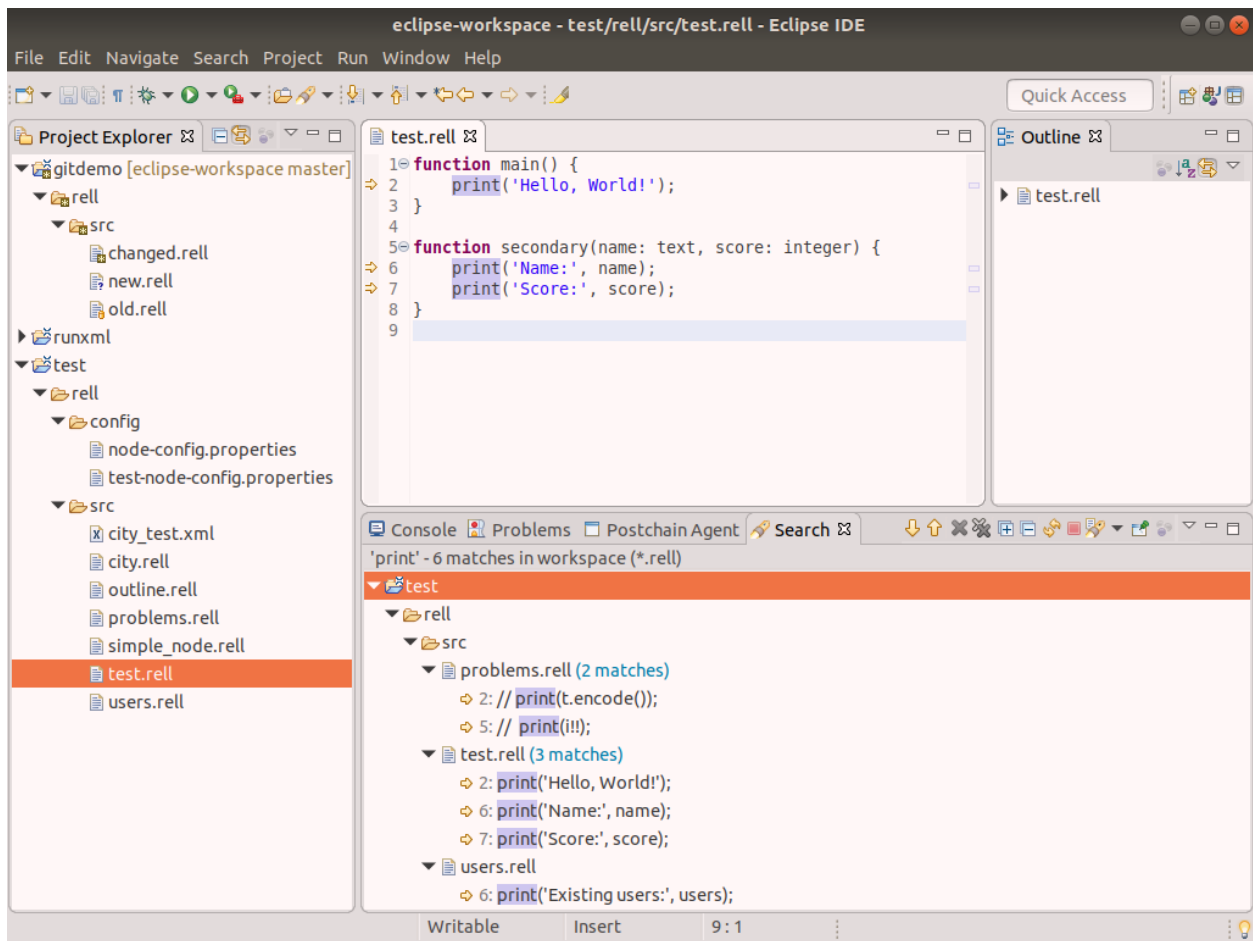
## Global Text Search

Press CTRL-H (H) to open the Search dialog. It allows to search for a string in all files in the IDE. Select the **File Search** tab, enter the text to search for and file name pattern.



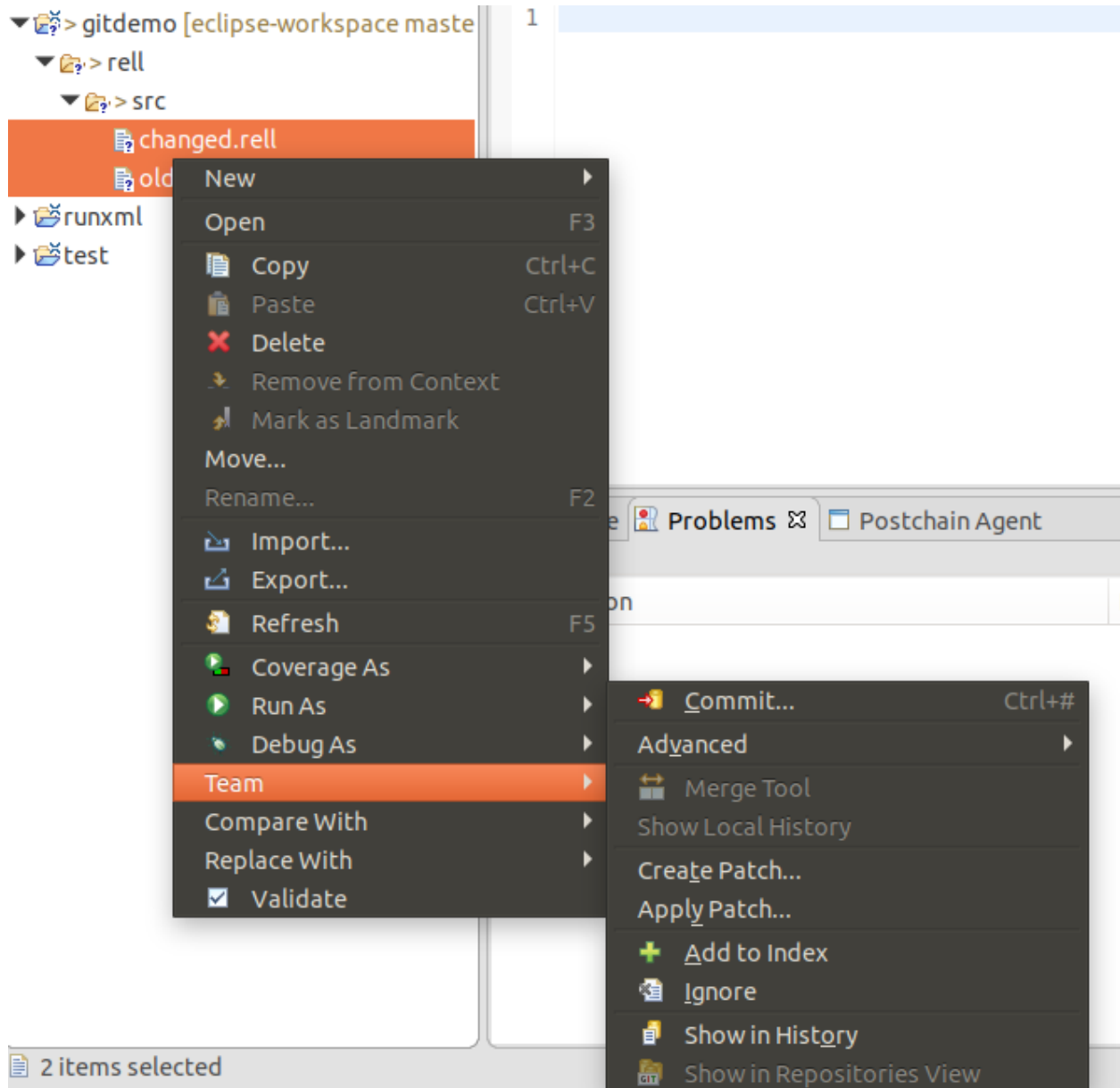


Results are shown in the Search view, as a tree.



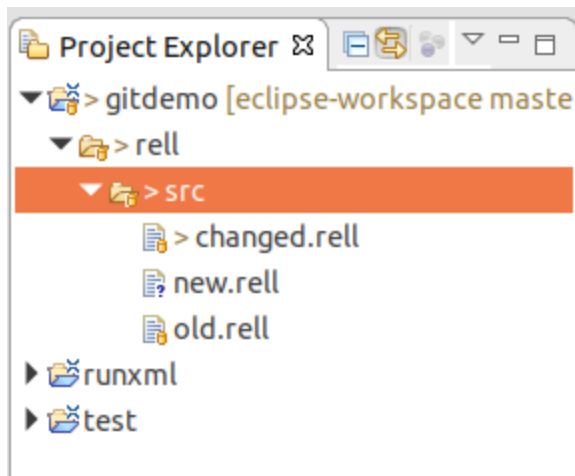
## Git

Git operations are available via a context menu: right-click on a file and choose **Team**.



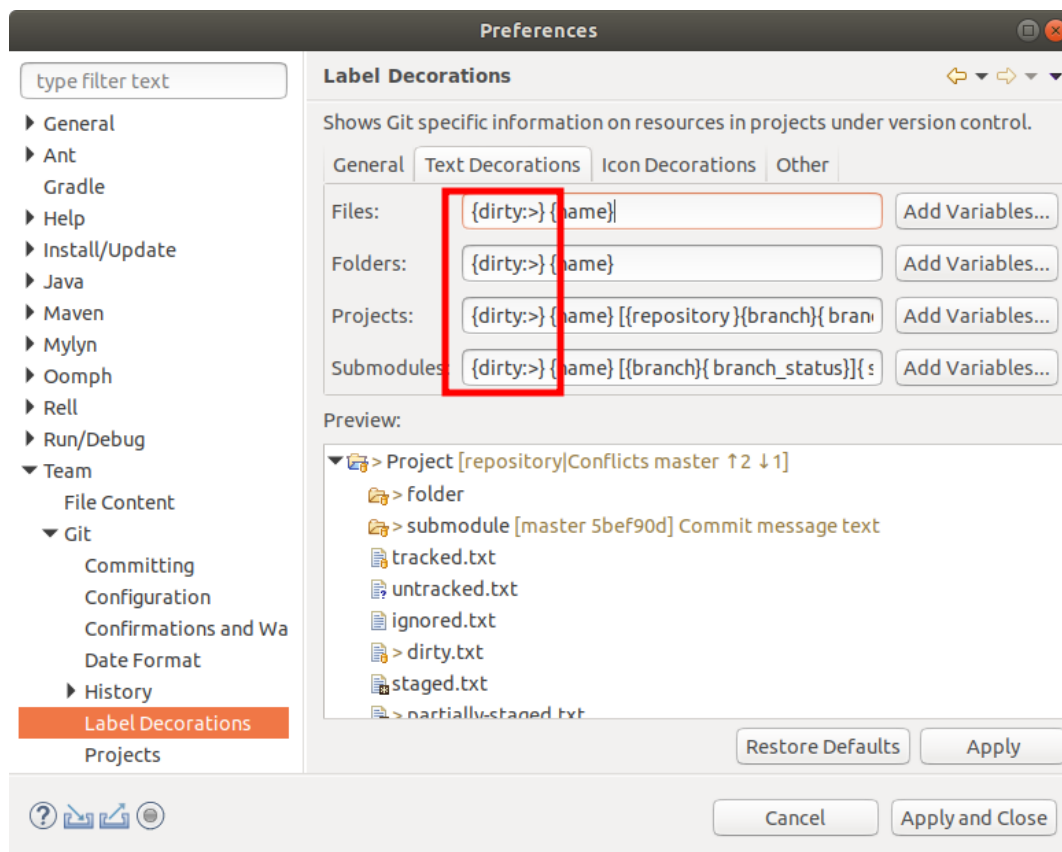
To commit file(s), click **Add to Index**, then **Commit...**

File icon in the Project Explorer indicates whether the file is a new, changed or an unmodified file.

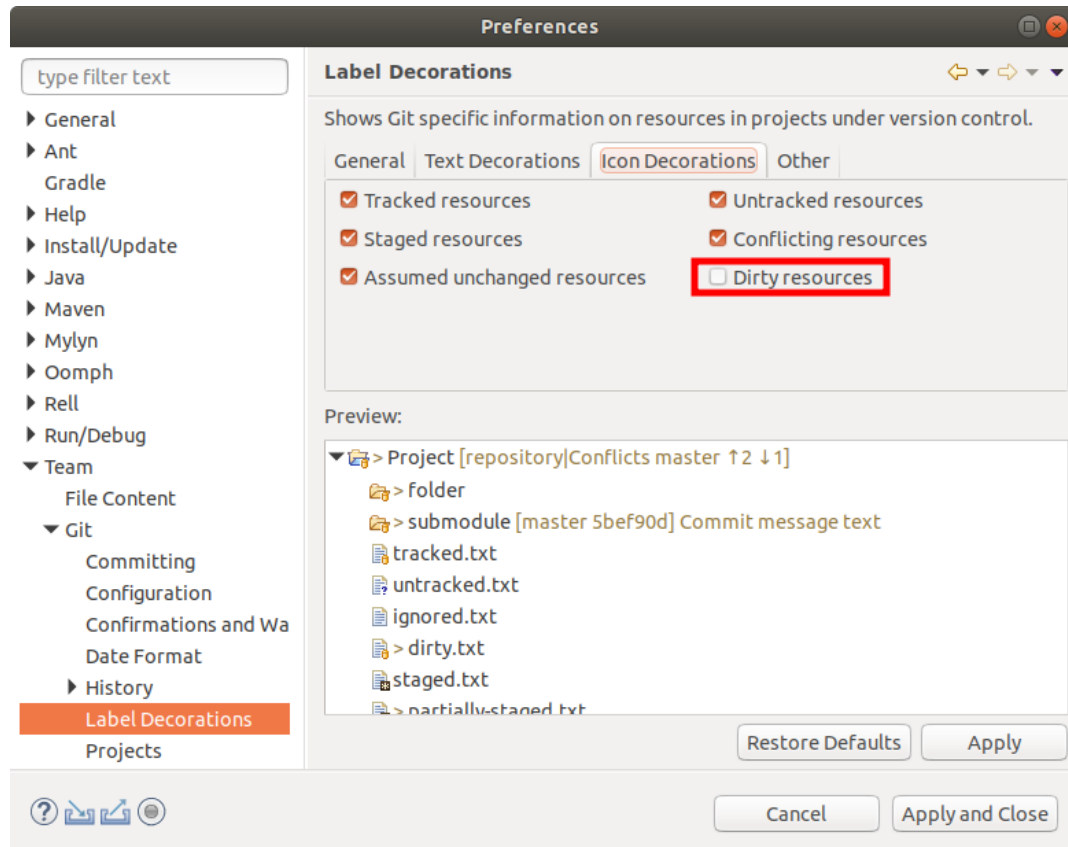


Changed files do not look very nice. To change the way how they are displayed, go to the **Window - Preferences** (macOS: **Eclipse - Preferences**) menu, then **Team - Git - Label Decorations**:

- on the **Text Decorations** tab: delete the “>” character in all text fields

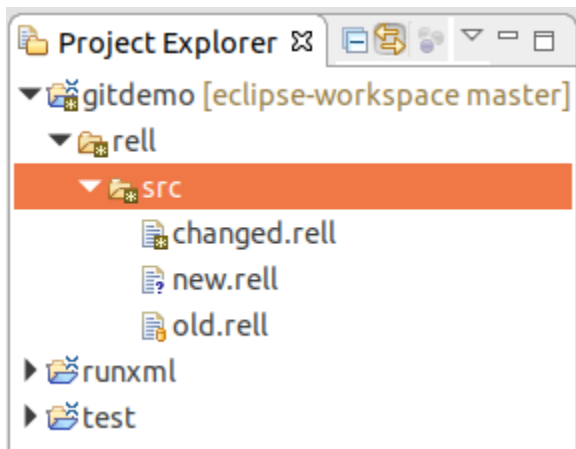


- on the **Icon Decorations** tab: check the **Dirty resources** checkbox



- click **Apply and Close**

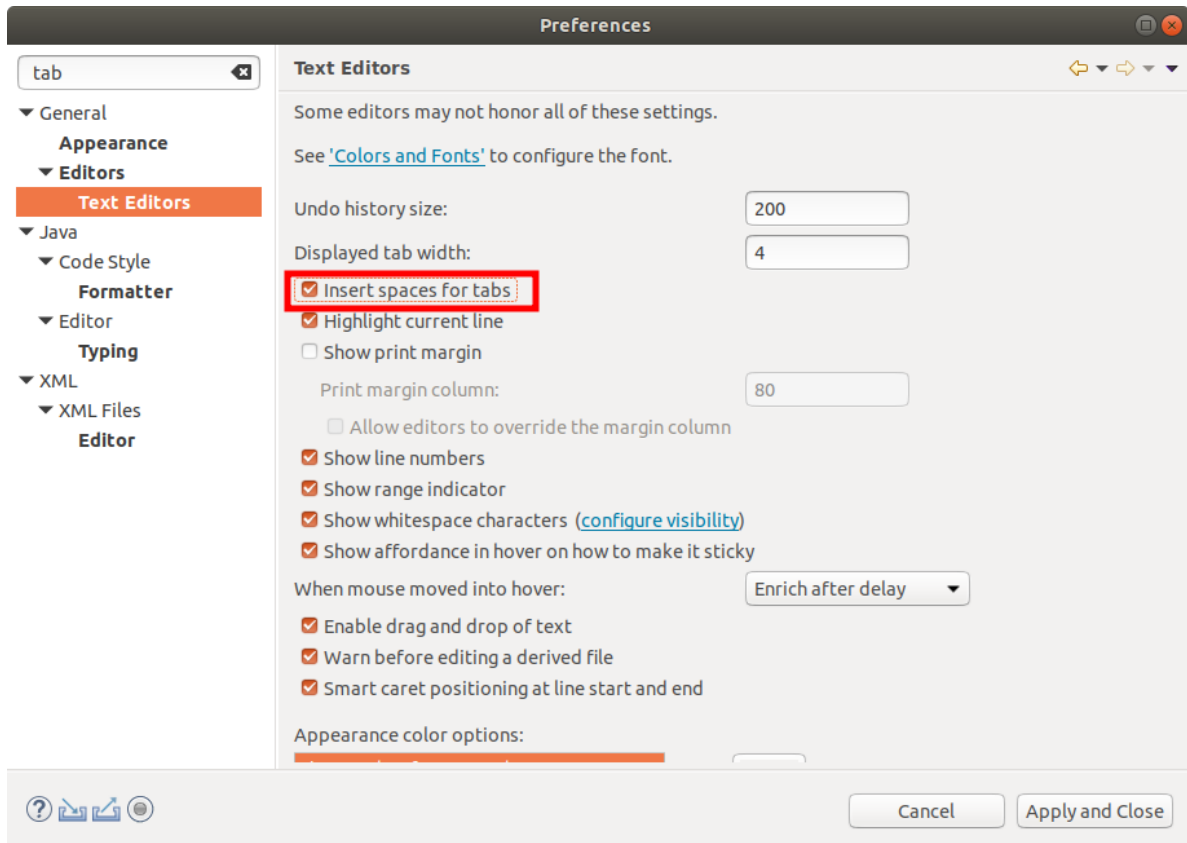
Now files look much better:



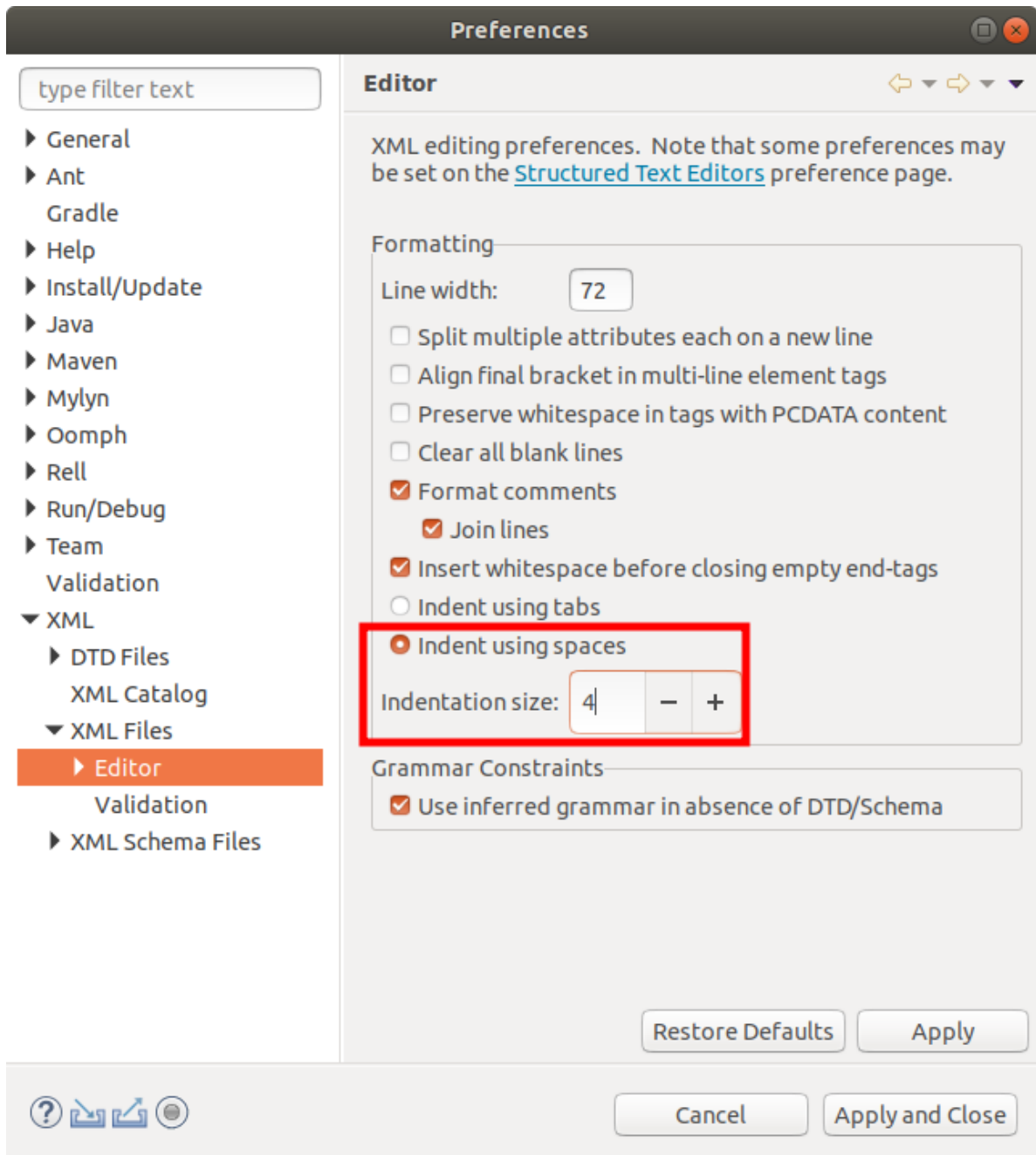
## Spaces vs. Tabs

Eclipse uses tabs instead of spaces by default. It is recommended to use spaces. A few settings have to be changed:

1. Open the Preferences dialog: menu **Window - Preferences** (macOS: **Eclipse - Preferences**).
2. Type “tabs” in the search box.
3. Go to **General - Editors - Text Editors** and check **Insert spaces for tabs**.

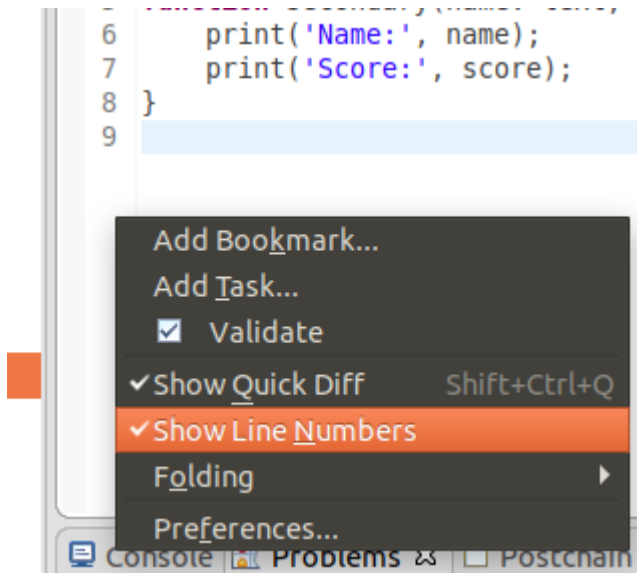


4. Go to **XML - XML Files - Editor**, select **Indent using spaces**, specify **Indentation size**: 4.

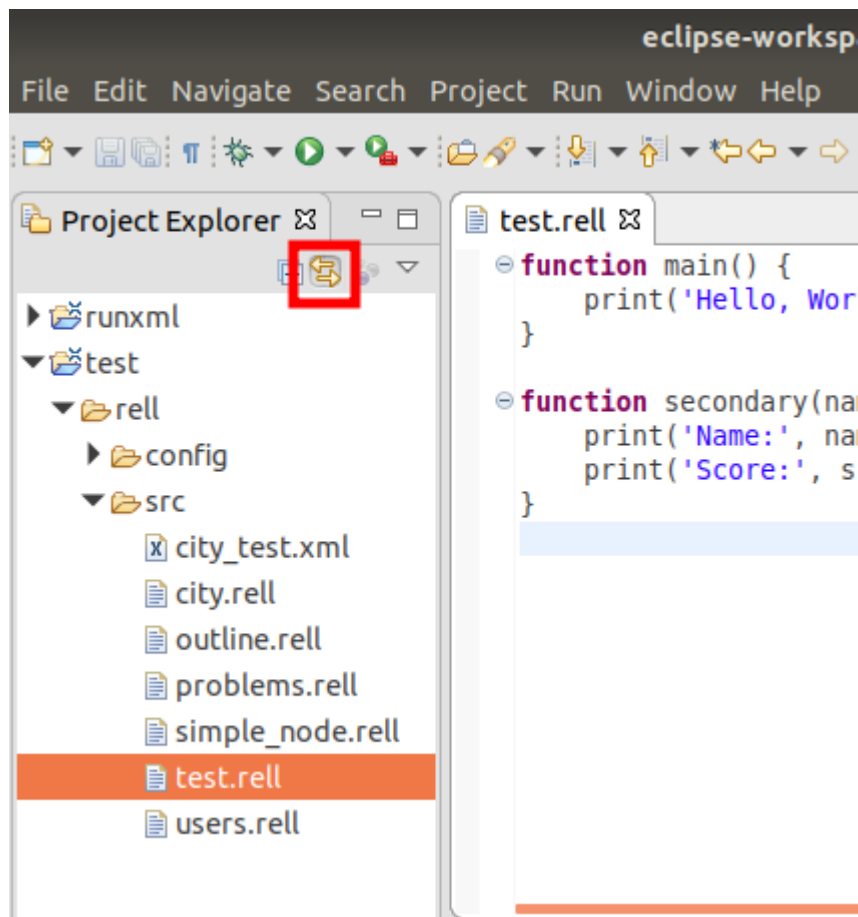


### Miscellaneous

- To show or hide line numbers: right-click on the left margin of an editor, click **Show Line Numbers**.

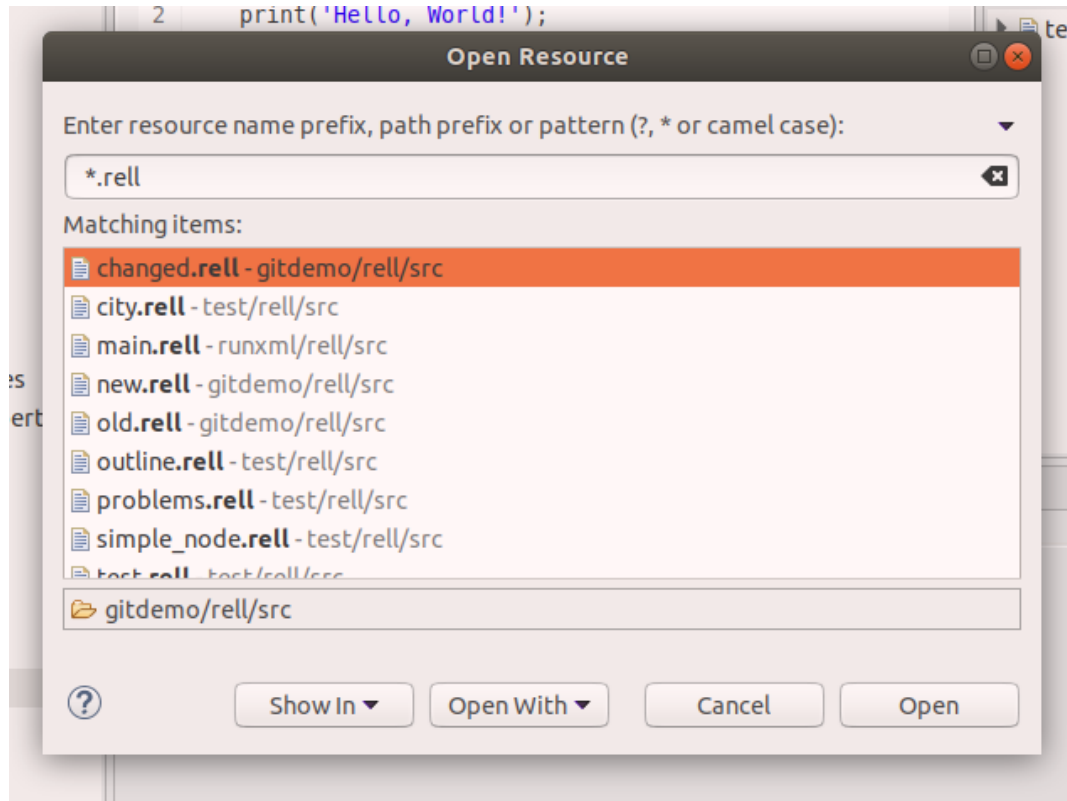


- To comment a code fragment, select it and press CTRL-/ (/).
- Activate the *Link with Editor* icon, and the IDE will automatically select a file in the Project Explorer when its editor is focused.

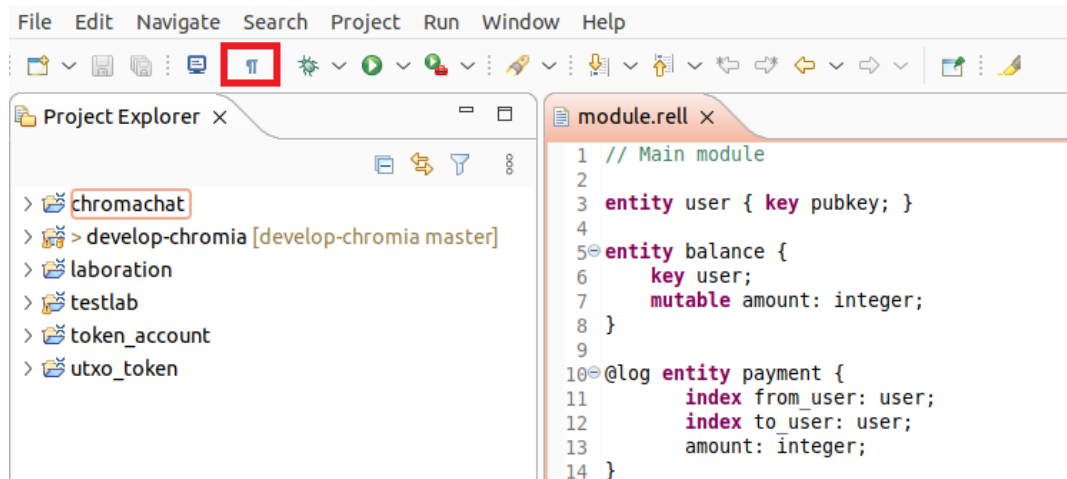


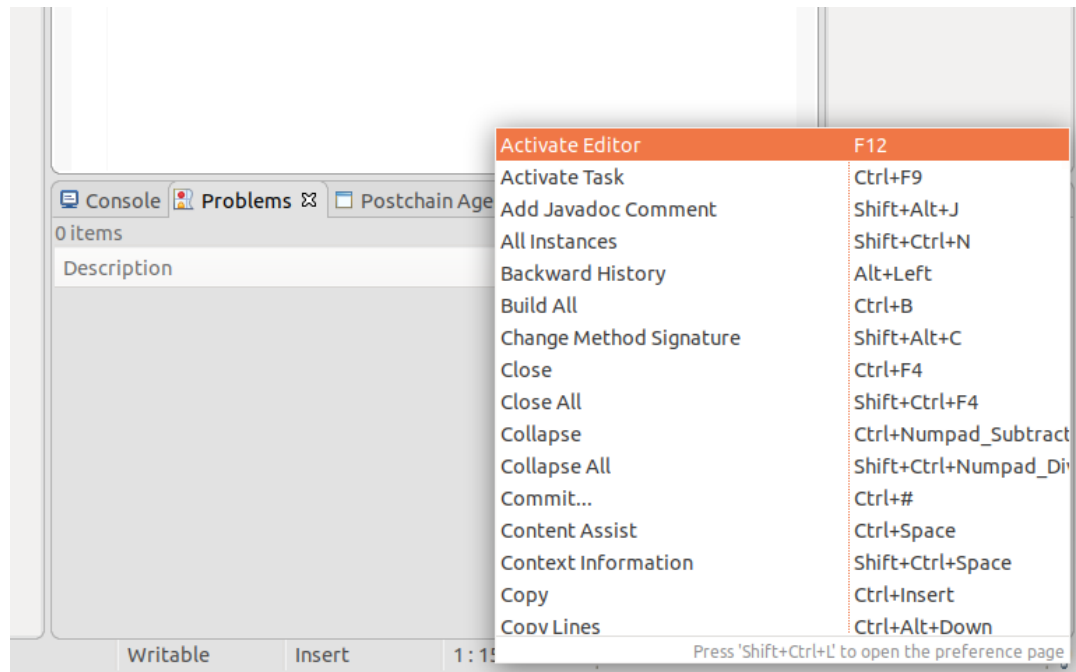
- CTRL-SHIFT-R (R) invokes the Open Resource dialog, which allows to search project files by name. Glob patterns (with \* and ?) are supported.





- The *Show Whitespace Characters* (paragraph) icon in the main toolbar allows to distinguish tabs from spaces.





## Keyboard shortcuts

Most useful keyboard shortcuts (subjectively):

Action	Linux/Windows	macOS
Run the file shown in the active editor	CTRL-F11	F11
Show the New wizard	CTRL-N	N
Show the New menu	ALT-SHIFT-N	N
Save the current file	CTRL-S	S
Close the active editor tab	CTRL-W	W
Close all editor tabs	CTRL-SHIFT-W	W
Quick outline	CTRL-O	O
Find text in the active editor	CTRL-F	F
Globally search for the selected text fragment	CTRL-ALT-G	G
Show global text search dialog	CTRL-H	^H
Find a file by name	CTRL-SHIFT-R	R
Go to a line number	CTRL-L	L
Go to a previous location	ALT-Left	←
Go to a next location (can be used after Alt-Left)	ALT-Right	→
Go to the last edit location	CTRL-Q	Q
Comment the selected code fragment with //	CTRL-/	/
Convert selected text to upper case	CTRL-SHIFT-X	X
Convert selected text to lower case	CTRL-SHIFT-Y	Y
Show the full list of keyboard short-cuts	CTRL-SHIFT-L	L

## 2.4 Example Projects

### 2.4.1 Chroma Chat

In this section we will write the ReII backend for a public chat dapp.

#### Requirements

The requirements we set are the following:

- There is one admin with an amount of tokens automatically assigned (say 1000000).
- The admin is the first person that registers themselves on the dapp.
- Any registered user can register a new user and transfer some tokens to them, after having paid 100 tokens to the admin as a fee.
- Users are identified by their public key.
- Channels are streams of messages belonging to the same topic, specified by the name of the channel (e.g. “showerthoughts”, where you can share thoughts you had in the shower).
- Registered users can create channels.
- When a new channel is created, only the creator is within the group. She can add any *existing* users. This operation costs 1 token.

## Entity definition

The structure of it will be:

```
entity user {
  key pubkey;
  key username: text;
}

entity channel {
  key name;
  admin: user;
}

entity channel_member {
  key channel, member: user;
}

entity message {
  key channel, timestamp;
  index posted_by: user;
  text;
}

entity balance {
  key user;
  mutable amount: integer;
}
```

Let’s analyse it:

### User

A user can be identified either by its pubkey or by its username. Both pubkey and username are key attributes and are therefore unique.

### Channel

Channels are identified by the name (which ideally reflects the topic of the channel itself) and the user who created it. Note that two channels cannot have the same name (key) and that a user can be admin of multiple channels.

## Message

One message has the text and reference of the user who sent it. Additionally, the channel and timestamp of publication is recorded. Note that key `channel`, `timestamp` means that only one message can be sent within a channel at given timestamp (but of course several messages on different channels can be recorded at single timestamp).

## Balance

This is kind of self explanatory: a user has an amount of tokens. Tokens can be spent (or more in general transferred), for this reason the field is marked as `mutable`.

## Operations

Operations are necessary when some data in the database is to be modified.

## Init

The module is initialized by performing the init operation. Here, an admin user is created with an account balance of 1000000. We don't want it to be possible to execute the operation a second time.

`require( (user@*{} limit 1).size() == 0 );` prevents that.

```
operation init (founder_pubkey: pubkey) {
  require( (user@*{} limit 1).size() == 0 );
  val founder = create user (founder_pubkey, "admin");
  create balance (founder, 1000000);
}
```

The operation receives a public key as input (note that it does not verify that signer of the transaction is the same specified in input field `founder_pubkey`, meaning you can specify a different public key).

## Transfer tokens (Function)

For convenience we create a function to transfer token from one user's balance to another's. We write it because we don't want to duplicate our checks and potentially create bugs.

```
function transfer_balance(from:user, to:user, amount:integer){
  require( balance@{from}.amount >= amount);
  update balance@{from} (amount -= amount);
  update balance@{to} (amount += amount);
}
```

We also add a `pay_fee` function that is a transfer from one user to the admin account:

```
function pay_fee (user, deduct_amount: integer) {
  if(user.username != 'admin'){
    transfer_balance(user, user@{.username == 'admin'}, deduct_amount);
  }
}
```

## Register a new user

As said, registered users should be allowed to add new users:

```
operation register_user (
  existing_user_pubkey: pubkey,
  new_user_pubkey: pubkey,
  new_user_username: text,
  transfer_amount: integer
) {
  require( op_context.is_signer(existing_user_pubkey) );
  val existing_user = user@{existing_user_pubkey};

  require( transfer_amount > 0 );

  val new_user = create_user (new_user_pubkey, new_user_username);
  pay_fee(existing_user, 100);

  create_balance (new_user, 0);
  transfer_balance(existing_user, new_user, transfer_amount);
}
```

Here we:

- Verify that the signer exists with `user@{existing_user_pubkey}`, which require exactly one result for the pubkey.
- Pay the fee of 100 tokens (transfer 100 tokens to ‘admin’ account)
- Then create the new user and transfer to them the specified positive amount of tokens.

---

**Note:** If at any point in the operation the conditions fail (for example, when the new username is already taken), the whole operation is rolled back and the transaction is rejected.

This is why we don’t need to check if the signer’s balance has `registration_cost + transfer_amount` tokens beforehand.

---

## Create a new channel

Registered users can create new channels. Given the public key and the name of the channel, we will verify that she is an actual registered user, transfer the fee, create the channel, and add that user as chat member.

```
operation create_channel ( admin_pubkey: pubkey, name) {
  require( op_context.is_signer(admin_pubkey) );
  val admin_usr = user@{admin_pubkey};
  pay_fee(admin_usr, 100);
  val channel = create_channel (admin_usr, name);
  create_channel_member (channel, admin_usr);
}
```

## Add user to channel

The admin of a channel (the one who created the channel) can add another user after having paid a fee of 1 token.

So we check once again that the signer is the `admin_pubkey` specified, we have the channel admin pay 1 token, and we add a new user to the channel via `channel_member`.

```
operation add_channel_member (admin_pubkey: pubkey, channel_name: name, member_
→username: text) {
    require( op_context.is_signer(admin_pubkey) );
    val admin_usr = user@{admin_pubkey};
    pay_fee(admin_usr, 1);
    val channel = channel@{channel_name, .admin==user@{admin_pubkey}};
    create channel_member (channel, member=user@{.username == member_username});
}
```

## Post a new message

People in a channel will love to share their opinions. They can do so with the `post_message` operation. The signer (`op_context.is_signer(pubkey)`) can post a message in the channel (`val channel = channel@{channel_name};`) if they are a member of the channel (`require( channel_member@? {channel, member} );`).

After the payment of 1 token fee, we add the new message to the channel:

```
operation post_message (channel_name: name, pubkey, message: text) {
    require( op_context.is_signer(pubkey) );
    val channel = channel@{channel_name};
    val member = user@{pubkey};
    require( channel_member@?{channel, member} );
    pay_fee(member, 1);
    create message (channel, member, text=message, op_context.last_block_time);
}
```

## Queries

It is useful to write data into a database in a distributed fashion, although writing would be meaningless without the ability to read.

### Query all channels where a user is registered

Getting the channels one user is registered into is simple, selecting from `channel_member` with the given user's public key.

```
query get_channels(pubkey):list<(name:text, admin: text)> {
    return channel_member@*{.member == user@{pubkey}} (name = .channel.name, admin = .
→channel.admin.username);
}
```

### Other simple queries

Likewise we can get the balance from one user.

```
query get_balance(pubkey) {
    return balance@{ user@{ pubkey } }.amount;
}
```

Retrieve messages sent in one channel sorted from the oldest to newest (sort .timestamp).

```
query get_last_messages(channel_name: name):list<(text:text, poster:text, ↵
↵timestamp:timestamp)> {
  return message@*{ channel@{channel_name} }
    ( .text, poster=.posted_by.username, @sort .timestamp );
}
```

### Run it

- Browse to <https://rellide-staging.chromia.dev>
- Create a new project
- Enter the above code in the code section (You can copy the full code from [here](#)).
- Click on Start Node (The green “Play” icon)

### Or

- Enter the above code in a new Eclipse Rell project
- Run it as a “Rell Postchain App” from the run.xml file

Congratulations! You should now have a running node.

### Client side

At this stage we should have a running node with your *freshly made* module.

What about interface it with a classy JS based application?

Well to do it we need the postchain-client npm package

```
npm i --save postchain-client
```

Lets open a new script in an editor of your liking and include the postchain client and crypto package.

```
const pcl = require('postchain-client');
const crypto = require('crypto');
```

Then we need to declare the address of the REST server (which is ran by the node, default is 7740) and the blockchain-RID of the blockchain and the number of sockets (5).

We then get an instance of GTX Client, via `gtxClient.createClient` and giving the rest object and blockchain-RID in input. Last parameters is an empty list of operation (this is needed if you don't use Rell language, in fact, you can also code a module with standard SQL or as a proper kotlin/java module).

```
// Check the node log on rellide-staging.chromia.dev to get node api url.
const nodeApiUrl = "https://rellide-staging.chromia.dev/node/XXXXX/";
const blockchainRID =
↵"78967baa4768cbcef11c508326ffb13a956689fcb6dc3ba17f4b895cbb1577a3"; // default RID ↵
↵on rellide-staging.chromia.dev
const rest = pcl.restClient.createRestClient(nodeApiUrl, blockchainRID, 5)
const gtx = pcl.gtxClient.createClient(
  rest,
```

(continues on next page)



(continued from previous page)

```

    Buffer.from(
      blockchainRID,
      'hex'
    ),
    []
  );

```

**Note:** If you are using Eclipse IDE, the configs should be:

```

const nodeApiUrl = "http://localhost:7740/"; //If using another port you can
↪specify it here
const blockchainRID =
↪"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"; //Blockchain
↪RID can be seen in the console window when starting a node

```

**Note:** If you are writing your script in the web IDE, you do not have to write configs as it is already included.

## Create and send a transaction with the init operation

First thing we probably want is to register and create the admin, we do so calling the `init` function.

```

function init(adminPubkey, adminPrivkey) {
  const rq = gtx.newTransaction([adminPubkey]);
  rq.addOperation('init', adminPubkey);
  rq.sign(adminPrivkey, adminPubkey);
  return rq.postAndWaitConfirmation();
}

```

The first thing we do is to declare a new transaction and that it will be signed by admin private key (we provide the public key, so the node can verify the veracity of transaction).

We add the operation called `init` and we pass as input argument the admin public key. We then sign the transaction with the private key (we specify the public key in order to correlate which private key refers to which public key in case of multiple signatures).

Finally we send the transaction to the node via the method `postAndWaitconfirmation` which returns a promise and resolves once it is confirmed.

Given the following keypair, we can create the admin.

```

const adminPUB = Buffer.from(
  '031b84c5567b126440995d3ed5aaba0565d71e1834604819ff9c17f5e9d5dd078f',
  'hex'
);
const adminPRIV = Buffer.from(
  '0101010101010101010101010101010101010101010101010101010101010101',
  'hex'
);
init(adminPUB, adminPRIV);

```

**Note:** In your own project, you might want to generate the keypair using `pcl.util.makeKeyPair()` instead:

```
const user = pcl.util.makeKeyPair();
const { pubKey, privKey } = user;
```

## Create other operations

We can also create a new channel, post a message, invite a user to dapp, invite a user in a channel

```
function createChannel(admin, channelName) {
  const pubKey = pcl.util.toBuffer(admin.pubKey);
  const privKey = pcl.util.toBuffer(admin.privKey);
  const rq = gtx.newTransaction([pubKey]);
  rq.addOperation("create_channel", pubKey, channelName);
  rq.sign(privKey, pubKey);
  return rq.postAndWaitConfirmation();
}

function postMessage(user, channelName, message) {
  const pubKey = pcl.util.toBuffer(user.pubKey);
  const privKey = pcl.util.toBuffer(user.privKey);
  const rq = gtx.newTransaction([pubKey]);
  rq.addOperation("nop", crypto.randomBytes(32));
  rq.addOperation("post_message", channelName, pubKey, message);
  rq.sign(privKey, pubKey);
  return rq.postAndWaitConfirmation();
}

function inviteUser(existingUser, newUserPubKey, startAmount) {
  const pubKey = pcl.util.toBuffer(existingUser.pubKey);
  const privKey = pcl.util.toBuffer(existingUser.privKey);
  const rq = gtx.newTransaction([pubKey]);
  rq.addOperation("register_user", pubKey, pcl.util.toBuffer(newUserPubKey),
  ↪parseInt(startAmount));
  rq.sign(privKey, pubKey);
  return rq.postAndWaitConfirmation();
}

function inviteUserToChat(existingUser, channel, newUserPubKey) {
  const pubKey = pcl.util.toBuffer(existingUser.pubKey);
  const privKey = pcl.util.toBuffer(existingUser.privKey);
  const rq = gtx.newTransaction([pubKey]);
  rq.addOperation("add_channel_member", pubKey, channel, pcl.util.
  ↪toBuffer(newUserPubKey));
  rq.sign(privKey, pubKey);
  return rq.postAndWaitConfirmation();
}
```

Although there is really nothing critical in these functions, there are few things worth noting:

- We expect public and private keys in hex format, and we convert them to Buffer with `pcl.util.toBuffer(admin.pubKey)`;
- In order to protect the system from replay attacks, the blockchain does not accept transactions which hash

is equal to an already existing transaction. This means that an user is not allowed to write the same message twice in a channel since if at day one he writes “hello” the transaction will be something like `rq.addOperation("post_message", the_channel, user_pub, "hello");`, when he will write ‘hello’ a second time the transaction will be the same and therefore rejected. To solve this problem, we add a “nop” operation with some random bytes via `rq.addOperation("nop", crypto.randomBytes(32));`, and create a different transaction hash.

---

**Important:** It is very important to remember this limitation imposed upon transactions. If your transaction is rejected with no obvious reason, chances are high that it is missing a “nop” operation.

---

## Querying the blockchain from the client side

Previously we wrote the queries on blockchain side. Now we need to query from the dapp. To do so we use the previously mentioned `postchain-client` package.

```
// Rell query, reported here for easy look up
// query get_balance(user_pubkey: text) {
//   return balance@{user@{byte_array(user_pubkey)}}.amount;
// }

function getBalance(user) {
  return gtx.query("get_balance", {
    user_pubkey: user.pubKey
  });
}
```

As you can see everything is contained into `gtx.query`: the first argument is the query name in the rell module, and the second argument is the name of the expected attribute in the query itself wrapped in an object. The name of the object is the one specified in module and the value, of course, the value we want to send. Please note that buffer values must before be converted into hexadecimal strings.

Other queries:

```
function getChannels(user) {
  return gtx.query("get_channels", {
    user_pubkey: user.pubKey
  });
}

function getMessages(channel) {
  return gtx.query("get_last_messages", {channel_name: channel});
}
```

## Conclusion

At this point, we have created a Rell backend for the public chat, and a javascript client to communicate with it.

We encourage you to extend this sample in anyway you like, by for example adding a user interface, or maybe by adding a “transfer” operation to send tokens to another user?

Or, if you are eager to see the application in its running state, we have implemented a simple UI for it at <https://bitbucket.org/chromawallet/chat-sample/src/master/>.

## 2.4.2 Account-based token system

Tokens are the bread & butter of blockchains, thus it is useful to demonstrate how a token system can be implemented in Rel. There are roughly two different implementation strategies:

- Account-based tokens which maintain an updatable balance for each account (which can be associated with a key or an address)
- UTXO-based ones (Bitcoin-style) deal with virtual “coins” which are minted and destroyed in transactions

This section details the account-based implementation. For an example of a UTXO based system see [UTXO-based token system](#).

A minimal implementation can look like this:

```
entity balance {
    key pubkey;
    mutable amount: integer;
}

operation transfer(from_pubkey: pubkey, to_pubkey: pubkey, xfer_amount: integer) {
    require( op_context.is_signer(from_pubkey) );
    require( xfer_amount > 0 );
    require( balance@{from_pubkey}.amount >= xfer_amount );
    update balance@{from_pubkey} (amount -= xfer_amount);
    update balance@{to_pubkey} (amount += xfer_amount);
}
```

There are a few items which should be highlighted in this code. First, let’s note that `balance@{from_pubkey}.amount` is simply a shorthand notation for `balance@{from_pubkey} (amount)`.

`update` relational operator combines a relational expression specifying objects to update with a form which specifies how to update their attributes. Attributes are updatable only if they are marked as `mutable`.

---

**Note:** We don’t need to worry about concurrency issues (i.e. that the balance can change after we checked it) because Rel applies operations within a single blockchain sequentially.

---

But this minimal implementation is not very useful, as there’s no mechanism for a wallet to identify payments it receives (without somehow scanning the blockchain, or asking the payer to share the transaction with the recipient). Other blockchain systems might resort to third-party tools and complex protocols to handle this (for example, the Electrum Bitcoin wallet connects to Electrum Servers which perform blockchain indexing). Rel-based blockchains can just use built-in indexing to keep track of payment history. For example, by using the additional `payment` class. We will be using the `log` annotation to add the transaction as an attribute to the entity. This can be used to timestamp the transaction, more about the `log` annotation and the transaction entity can be found under the system library chapter. To make things more efficient, we also wrap `pubkey` into `user` class, thus getting:

```
entity user { key pubkey; }

entity balance {
    key user;
    mutable amount: integer;
}

@log entity payment {
    index from_user: user;
    index to_user: user;
    amount: integer;
}
```

(continues on next page)

(continued from previous page)

```

        timestamp;
    }

    operation transfer(from_pubkey: pubkey, to_pubkey: pubkey, xfer_amount: integer) {
        require( op_context.is_signer(from_pubkey) );
        require( xfer_amount > 0 );
        val from_user = user@{from_pubkey};
        val to_user = user@{to_pubkey};
        require( balance@{from_user}.amount >= xfer_amount );
        update balance@{from_user} (amount -= xfer_amount);
        update balance@{to_user} (amount += xfer_amount);
        create payment (
            from_user,
            to_user,
            amount=xfer_amount);
    }

```

**Note:** In `create payment (from_user, to_user, ...)` Rell can figure out matching attributes from names of local variables as they match exactly. It is often the case that you can use the same name for the same concept.)

The example above can be easily extended to support multiple types of tokens. For example:

```

entity asset { key asset_code; }

entity balance {
    key user, asset;
    mutable amount: integer;
}

```

Here we use a composite key to keep track of the balance for each `(user, asset)` pair.

## Client Side API

Lets see how we would call this transfer in the front-end. First of all, we need to initialize a user with some starting money. We also need a way to add more users to the network. So before we start writing the front-end we add an init function to our Rell module and also a register user function.

The init function:

```

operation init (founder_pubkey: pubkey) {

    require( (user@*{ } limit 1).size() == 0 );
    val founder = create user(founder_pubkey);
    create balance (founder, 1000000);
}

```

The register user function:

```

operation register_user (
    existing_user_pubkey: byte_array,
    new_user_pubkey: byte_array
) {
    require( op_context.is_signer(existing_user_pubkey) );
}

```

(continues on next page)

(continued from previous page)

```
val existing_user = user@{existing_user_pubkey};
val new_user = create user (new_user_pubkey);
create balance (new_user, 0);
}
```

Now you can start writing a front-end in nodeJS. If you need a refresher on the installation, check out the “Client Side” chapter in Reli basics. We start by adding the postchain package and start an instance of a GTX client.

```
const pcl = require('postchain-client');
const nodeApiUrl = "https://rellide-staging.chromia.dev/node/XXXXX/"; //Fill this
    ↳ url with where your node is.
const blockchainRID =
    ↳ "78967baa4768cbcef11c508326fffb13a956689fcb6dc3ba17f4b895cbb1577a3"; // default RID
    ↳ on rellide-staging.chromia.dev
const rest = pcl.restClient.createRestClient(nodeApiUrl, blockchainRID, 5)
const gtx = pcl.gtxClient.createClient(
    rest,
    Buffer.from(
        blockchainRID,
        'hex'
    ),
    []
);
```

Now we have all we need to start sending transactions to our backend. We start by defining a function that sends a transaction with the init operation inside.

```
async function initialize(admin){
    const adminPubKey = pcl.util.toBuffer(admin.pubKey);
    const tx = gtx.newTransaction([admin.pubKey]);
    tx.addOperation("init", adminPubKey);
    tx.sign(admin.privKey, admin.pubKey);
    await tx.postAndWaitConfirmation();
}
```

Now we can write out the function for registering a new user:

```
async function registerUser(newUser, oldUser){
    const newUserPubKey = pcl.util.toBuffer(newUser.pubKey);
    const oldUserPubKey = pcl.util.toBuffer(oldUser.pubKey);
    const oldUserPrivKey = pcl.util.toBuffer(oldUser.privKey);
    const tx = gtx.newTransaction([oldUserPubKey]);
    tx.addOperation("register_user", oldUserPubKey, newUserPubKey);
    tx.sign(oldUserPrivKey, oldUserPubKey);
    await tx.postAndWaitConfirmation();
}
```

And lastly, the transfer function:

```
async function transferBalance(fromUser, toUser, amount) {
    const fromUserPubKey = pcl.util.toBuffer(fromUser.pubKey);
    const fromUserPrivKey = pcl.util.toBuffer(fromUser.privKey);
    const toUserPubKey = pcl.util.toBuffer(toUser.pubKey);
    const tx = gtx.newTransaction([fromUserPubKey]);
    tx.addOperation("transfer_balance", fromUserPubKey, toUserPubKey, amount);
    tx.sign(fromUserPrivKey, fromUserPubKey);
}
```

(continues on next page)

(continued from previous page)

```

    await tx.postAndWaitConfirmation();
}

```

### 2.4.3 UTXO-based token system

As an exercise, we can also implement a Bitcoin-style token system.

We first define an unspent transaction output structure:

```

entity utxo {
    pubkey;
    amount: integer;
}

```

Then define the transfer operation that roughly follows Bitcoin transaction structure – it has a list of inputs and outputs:

```

operation transfer (inputs: list<utxo>, output_pubkeys: list<pubkey>, output_amounts:
↳list<integer>) {
    var input_sum = 0;
    for (an_utxo in inputs) {
        require(op_context.is_signer(an_utxo.pubkey));
        input_sum += an_utxo.amount;
        delete utxo@{utxo == an_utxo};
    }
    var output_sum = 0;
    require(output_pubkeys.size() == output_amounts.size());
    for (out_index in range(output_pubkeys.size())) {
        output_sum += output_amounts[out_index];
        create utxo (output_pubkeys[out_index],
                    output_amounts[out_index]);
    }
    require(output_sum <= input_sum);
}

```

There are quite a lot of new constructs used in this example:

- `list<...>` is, obviously, a collection. Besides lists, Rell also supports set and map
- `in list<utxo>` `utxo` object references are physically implemented using integer identifiers which are used internally
- `an_utxo.pubkey` accesses an attribute of an object, which is a database query identical to `utxo@{utxo==an_utxo} (pubkey)`
- variable type is automatically inferred from expression used for initialization. One can also write it like `var output_sum : integer = 0;`
- `delete` operation accepts a relational expression which identifies object(s)
- `.size()` method can be used get the size of a collection
- `for (... in ...)` works both for collections and for ranges of integer values
- `[]` is used to refer to an element of a collection

**Note:** A front-end client together with a similar UTXO implementation to the example above can be found [here](#).

Note that we perform checks as we go. This is OK because Rel is transactional: if a requirement fails or an error is generated, the whole operation (in fact, the whole transaction) is rolled back. Rel is typically used with a GTX transaction format which supports multiple signers and multiple operations per transaction. Thus it can easily support Bitcoin-style multi-input transactions, atomic token swaps, multi-sig etc.

Now a bit about `delete` operator. Isn't it strange to enable deletion of data from a blockchain?!

Here we aren't deleting data "from a blockchain", we are removing entries from *the current blockchain state*. This is exactly how it works in a Bitcoin node – once entries in an unspent transaction output set are spent, they are deleted. A typical Bitcoin node doesn't keep track of spent transaction outputs.

A system based on Rel (e.g. Postchain or Chromia) works in exactly the same way: raw information about transactions and operations is preserved in a blockchain. The database contains both raw blockchain transactions and processed current state. The current state is what a Rel programmer can work with: he is allowed to do destructive updates and delete entries. These operations do not affect the raw blockchain.

## 2.5 Language Features

### 2.5.1 Types

#### Table of Contents

- *Types*
  - *Simple types*
    - \* *boolean*
    - \* *integer*
    - \* *decimal*
    - \* *text*
    - \* *byte\_array*
    - \* *rowid*
    - \* *json*
    - \* *unit*
    - \* *null*
    - \* *Simple type aliases*
  - *Complex types*
    - \* *entity*
    - \* *struct*
    - \* *enum*
    - \* *T? - nullable type*
    - \* *tuple*
    - \* *range*
    - \* *gtv*



- *Collection types*
  - \* *list<T>*
  - \* *set<T>*
  - \* *map<K,V>*
- *Virtual types*
  - \* *virtual<list<T>>*
  - \* *virtual<set<T>>*
  - \* *virtual<map<K,V>>*
  - \* *virtual<struct>*
- *Subtypes*

## Simple types

### boolean

```
val using_rell = true;
if (using_rell) print("Awesome!");
```

### integer

```
val user_age : integer = 26;
```

`integer.MIN_VALUE` = minimum value ( $-2^{63}$ )

`integer.MAX_VALUE` = maximum value ( $2^{63}-1$ )

`integer(s: text, radix: integer = 10)` - parse a signed string representation of an integer, fail if invalid

`integer(decimal): integer` - converts a decimal to an integer, rounding towards 0 (5.99 becomes 5, -5.99 becomes -5), throws an exception if the resulting value is out of range

`integer.from_text(s: text, radix: integer = 10): integer` - same as `integer(text, integer)`

`integer.from_hex(text): integer` - parse an unsigned HEX representation

`.abs(): integer` - absolute value

`.max(integer): integer` - maximum of two values

`.max(decimal): decimal` - maximum of two values (converts this integer to decimal)

`.min(integer): integer` - minimum of two values

`.min(decimal): decimal` - minimum of two values (converts this integer to decimal)

`.to_text(radix: integer = 10)` - convert to a signed string representation

`.to_hex(): text` - convert to an unsigned HEX representation

`.sign()`: integer - returns -1, 0 or 1 depending on the sign

## decimal

Represent a real number.

```
val approx_pi : decimal = 3.14159;
val scientific_value : decimal = 55.77e-5;
```

It is not a normal floating-point type found in many other languages (like `float` and `double` in C/C++/Java):

- `decimal` type is accurate when working with numbers within its range. All decimal numbers (results of decimal operations) are implicitly rounded to 20 decimal places. For instance, `decimal('1E-20')` returns a non-zero, while `decimal('1E-21')` returns a zero value.
- Numbers are stored in a decimal form, not in a binary form, so conversions to and from a string are lossless (except when rounding occurs if there are more than 20 digits after the point).
- Floating-point types allow to store much smaller numbers, like `1E-300`; `decimal` can only store `1E-20`, but not a smaller nonzero number.
- Operations on decimal numbers may be considerably slower than integer operations (at least 10 times slower for same integer numbers).
- Large decimal numbers may require a lot of space: ~0.41 bytes per decimal digit (~54KiB for `1E+131071`) in memory and ~0.5 bytes per digit in a database.
- Internally, the type `java.lang.BigDecimal` is used in the interpreter, and `NUMERIC` in SQL.
- In the code one can use decimal literals:

```
123.456
0.123
.456
33E+10
55.77e-5
```

Such numbers have `decimal` type. Simple numbers without a decimal point and exponent, like `12345`, have `integer` type.

`decimal.PRECISION`: integer - the maximum number of decimal digits (`131072 + 20`)

`decimal.SCALE`: integer - the maximum number of decimal digits after the decimal point (`20`)

`decimal.INT_DIGITS`: integer - the maximum number of digits before the decimal point (`131072`)

`decimal.MIN_VALUE`: decimal - the smallest nonzero absolute value that a `decimal` can store

`decimal.MAX_VALUE`: decimal - the largest value that can be stored in a `decimal` (`1E+131072 - 1`)

`decimal(integer)`: decimal - converts integer to decimal

`decimal(text)`: decimal - converts a text representation of a number to decimal

`.abs()`: decimal - absolute value

`.ceil()`: decimal - ceiling value: rounds 1.0 to 1.0, 1.00001 to 2.0, -1.99999 to -1.0, etc.

`.floor()`: decimal - floor value: rounds 1.0 to 1.0, 1.9999 to 1.0, -1.0001 to -2.0, etc.

`.min(decimal)`: decimal - minimum of two values

`.max(decimal)`: decimal - maximum of two values

`.round(scale: integer = 0): decimal` - rounds to a specific number of decimal places, to a closer value

- Example: `round(2.49) = 2.0`, `round(2.50) = 3.0`, `round(0.12345, 3) = 0.123`. Negative scales are allowed too: `round(12345, -3) = 12000`.

`.sign(): integer` - returns -1, 0 or 1 depending on the sign

`.to_integer(): integer` - converts a decimal to an integer, rounding towards 0

- Example: (5.99 becomes 5, -5.99 becomes -5)

`.to_text(scientific: boolean = false): text`

## text

Textual value. Same as string type in some other languages.

```
val placeholder = "Lorem ipsum donor sit amet";
print(placeholder.size()); // 26
print(placeholder.empty()); // false
```

`text.from_bytes(byte_array, ignore_invalid: boolean = false)` - if `ignore_invalid` is false, throws an exception when the byte array is not a valid UTF-8 encoded string, otherwise replaces invalid characters with a placeholder.

`.empty(): boolean` - returns true if the text is empty, otherwise returns false

`.size(): integer` - returns the number of characters.

`.compare_to(text): integer` - returns 0 if texts match, otherwise a positive or negative value

`.starts_with(text): boolean` - returns true if it starts with the input text, otherwise returns false

`.ends_with(text): boolean` - returns true if it ends with the input text, otherwise returns false

`.contains(text): boolean` - return true if contains the given substring, otherwise returns false

`.index_of(text): integer` - returns position of input text and -1 if substring is not found

`.index_of(text, integer): integer` - same as `.index_of(text)` but starting search from given position

`.last_index_of(text[, start: integer]): integer` - returns -1 if substring is not found (as in Java)

`.sub(start: integer[, end: integer]): text` - get a substring (start-inclusive, end-exclusive)

`.replace(old: text, new: text)` - replaces substring with new text

`.upper_case(): text` - converts the text to uppercase letters

`.lower_case(): text` - converts the text to lowercase letters

`.split(text): list<text>` - strictly split by a separator (not a regular expression)

`.trim(): text` - remove leading and trailing whitespace

`.matches(text): boolean` - returns true if it matches a regular expression

`.to_bytes(): byte_array` - convert to a UTF-8 encoded byte array

`.char_at(integer): integer` - get a 16-bit code of a character

`.format(...)` - formats a string (as in Java):

- 'My name is <%s>'.format('Bob') - returns 'My name is <Bob>'

Most of these text functions can be used within at-expressions where they will be translated to their SQL equivalents.

Special operators:

- + : concatenation
- [] : character access (returns single-character text)

## byte\_array

```
val user_pubkey : byte_array = x
  ↳ "0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15";
print(user_pubkey.to_base64()); //A3NZmmHMazvAKnjDQxPhc3rpz9Vrm7JDYlQ31Gnv3zsV
```

byte\_array(text) - creates a byte\_array from a HEX string, e.g. '1234abcd'

byte\_array.from\_hex(text): byte\_array - same as byte\_array(text)

byte\_array.from\_base64(text): byte\_array - creates a byte\_array from a Base64 string

byte\_array.from\_list(list<integer>): byte\_array - converts list to bytearray; values must be 0 - 255

.empty(): boolean - returns true if the text is empty, otherwise returns false

.size(): integer - returns the number of characters.

.sub(start: integer[, end: integer]): byte\_array - sub-array (start-inclusive, end-exclusive)

.to\_hex(): text - returns a HEX representation of the byte array, e.g. '1234abcd'

.to\_base64(): text - returns a Base64 representation of the byte array

.to\_list(): list<integer> - list of values 0 - 255

.sha256(): byte\_array - returns the sha256 digest as a byte\_array

Most of these byte array functions can be used within at-expressions where they will be translated to their SQL equivalents.

Special operators:

- + : concatenation
- [] : element access

## rowid

Primary key of a database record, 64-bit integer, supports only comparison operations

## json

Stored in Postgres as JSON type, and can be parsed to text;

```
val json_text = '{ "name": "Alice" }';
val json_value: json = json(json_text);
print(json_value);
```

`json(text)` - create a json value from a string; fails if not a valid JSON string

`.to_text()` : `text` - convert to string

## unit

No value; cannot be used explicitly. Equivalent to *unit* type in Kotlin.

## null

Type of `null` expression; cannot be used explicitly

## Simple type aliases

- `pubkey = byte_array`
- `name = text`
- `timestamp = integer`
- `tuid = text`

## Complex types

### entity

The entity represents a database object. An example to visualize how relations between objects are maintained can be seen below:

```
entity user {
    key pubkey;
    index name;
}
```

```
entity address {
    key pubkey;
    street: text;
}
```

```
entity residence {
    key user, address;
}

//both queries will result lists of records, as it is a many-to-many relationships
//they might be of different sizes. The constraint is that the composite of the two_
↳keys are unique.
function get_addresses(user): list<address> = residence @* { user }.address;
function get_users(address): list<user> = residence @* { address }.user;
```

is many-to-many relation, i.e. user can have more than one residence. Many-to-one can be implemented by changing key:

```
entity residence {
    key user;
    index address;
}
//There is a unique constraint on the user key.
function get_addresses(user): list<address> = residence @* { user }.address; //will
↳only return zero or one addresses
function get_users(address): list<user> = residence @* { address }.user; // will
↳return zero to n users
```

And for completeness, one-to-many would be reverse of many-to-one:

```
entity residence {
    index user;
    key address;
}
```

It also works without making address an entity, e.g.:

```
entity user { key name; }

// allow user to have many tags
entity user_tag {
    key user, tag: text;
}
```

and one-to-one (optional) as:

```
entity residence {
    key user;
    key address; //make sure that address can be claimed by at most one user
}
There is a unique constraint on the user key and the address key.
function get_addresses(user): list<address> = residence @* { user }.address; //will
↳only return zero or one addresses
function get_users(address): list<user> = residence @* { address }.user; // will only
↳return zero to one users
```

## struct

A struct is similar to an entity, but its instances exist in memory, not in a database.

```
struct user {
    name: text;
    address: text;
    mutable balance: integer = 0;
}
```

Functions available for all struct types:

`T.from_bytes(byte_array):` `T` - decode from a binary-encoded gtv (same as `T.from_gtv(gtv.from_bytes(x))`)

`T.from_gtv(gtv):` `T` - decode from a gtv

`T.from_gtv_pretty(gtv):` `T` - decode from a pretty-encoded gtv

`.to_bytes():` `byte_array` - encode in binary format (same as `.to_gtv().to_bytes()`)

```
.to_gtv() :   gtv - convert to a gtv
.to_gtv_pretty() :   gtv - convert to a pretty gtv
```

## enum

```
enum account_type {
    single_key_account,
    multi_sig_account
}

entity account {
    key id: byte_array;
    mutable account_type;
    mutable args: byte_array;
}
```

Assuming T is an enum type:

```
T.values() :   list<T> - returns all values of the enum, in the order of declaration
T.value(text) :   T - finds a value by name, throws an exception if not found
T.value(integer) :   T - finds a value by index, throws an exception if not found
```

### Enum value properties

```
.name:   text - the name of the enum value
.value:  integer - the numeric value (index) associated with the enum value
```

## T? - nullable type

```
val nonexistent_user = user @? { .name == "Nonexistent Name" };
require_not_empty(nonexistent_user); // Throws exception because user doesn't exists
```

- Entity attributes cannot be nullable.
- Can be used with almost any type (except nullable, unit, null).
- Nullable nullable (T?? is not allowed).
- Normal operations of the underlying type cannot be applied directly.
- Supports ?:, ?., and !! operators (like in Kotlin).

Compatibility with other types:

- Can assign a value of type T to a variable of type T?, but not the other way round.
- Can assign null to a variable of type T?, but not to a variable of type T.
- Can assign a value of type (T) (tuple) to a variable of type (T?).
- Cannot assign a value of type list<T> to a variable of type list<T?>.

Allowed operations:

- Null comparison: x == null, x != null.
- ?? - null check operator: x?? is equivalent to x != null

- `!!` - null assertion operator: `x!!` returns value of `x` if `x` is not null, otherwise throws an exception
- `?:` - Elvis operator: `x ? y` means `x` if `x` is not null, otherwise `y`
- `?.` - safe access: `x?.y` results in `x.y` if `x` is not null and null otherwise; similarly, `x?.y()` either evaluates and returns `x.y()` or returns null
- `require(x)`, `require_not_empty(x)`: throws an exception if `x` is null, otherwise returns value of `x`

Examples:

```
function f(): integer? { ... }

val x: integer? = f(); // type of "x" is "integer?"
val y = x;             // type of "y" is "integer?"

val i = y!!;           // type of "i" is "integer"
val j = require(y);    // type of "j" is "integer"

val a = y ? 456;       // type of "a" is "integer"
val b = y ? null;      // type of "b" is "integer?"

val p = y!!;           // type of "p" is "integer"
val q = y?.to_hex();   // type of "q" is "text?"

if (x != null) {
    val u = x;          // type of "u" is "integer" - smart cast is applied to "x"
} else {
    val v = x;          // type of "v" is "integer?"
}
```

## tuple

Examples:

- `val single_number : (integer) = (16,)` - one value
- `val invalid_tuple = (789)` - *not* a tuple (no comma)
- `val user_tuple: (integer, text) = (26, "Bob")` - two values
- `val named_tuple : (x: integer, y: integer) = (32, 26)` - named fields (can be accessed as `named_tuple.x`, `named_tuple.y`)
- `(integer, (text, boolean))` - nested tuple

Tuple types are compatible only if names and types of fields are the same:

- `(x:integer, y:integer)` and `(a:integer,b:integer)` are not compatible.
- `(x:integer, y:integer)` and `(integer, integer)` are not compatible.

Reading tuple fields:

- `t[0]`, `t[1]` - by index
- `t.a`, `t.b` - by name (for named fields)

Unpacking tuples:

```
val t = (123, 'Hello');
val (n, s) = t;           // n = 123, s = 'Hello'
```



Works for arbitrarily nested tuples:

```
val (n, (p, (x, y), q)) = calculate();
```

Special symbol `_` is used to ignore a tuple element:

```
val (_, s) = (123, 'Hello'); // s = 'Hello'
```

Variable types can be specified explicitly:

```
val (n: integer, s: text) = (123, 'Hello');
```

Unpacking can be used in a loop:

```
val l: list<(integer, text)> = get_tuples();
for ((x, y) in l) {
    print(x, y);
}
```

## range

Can be used in `for` statement:

```
for(count in range(10)){
    print(count); // prints out 0 to 9
}
```

`range(start: integer = 0, end: integer, step: integer = 1)` - start-inclusive, end-exclusive (as in Python):

- `range(10)` - a range from 0 (inclusive) to 10 (exclusive)
- `range(5, 10)` - from 5 to 10
- `range(5, 15, 4)` - from 5 to 15 with step 4, i. e. `[5, 9, 13]`
- `range(10, 5, -1)` - produces `[10, 9, 8, 7, 6]`
- `range(10, 5, -3)` - produces `[10, 7]`

Special operators:

- `in` - returns `true` if the value is in the range (taking `step` into account)

## gtv

A type used to represent encoded arguments and results of remote operation and query calls. It may be a simple value (integer, string, byte array), an array of values or a string-keyed dictionary.

Some Rell types are not Gtv-compatible. Values of such types cannot be converted to/from `gtv`, and the types cannot be used as types of operation/query parameters or result.

### Rules of Gtv-compatibility

- `range` is not Gtv-compatible
- a complex type is not Gtv-compatible if a type of its component is not Gtv-compatible

```
gtv.from_json(text): gtv - decode a gtv from a JSON string
gtv.from_json(json): gtv - decode a gtv from a json value
gtv.from_bytes(byte_array): gtv - decode a gtv from a binary-encoded form
.to_json(): json - convert to JSON
.to_bytes(): byte_array - convert to bytes
.hash(): byte_array - returns a cryptographic hash of the value
```

### gtv-related functions

Functions available for all Gtv-compatible types:

```
T.from_gtv(gtv): T - decode from a gtv
T.from_gtv_pretty(gtv): T - decode from a pretty-encoded gtv
.to_gtv(): gtv - convert to a gtv
null.to_gtv() - Returns the gtv equivalent of null
.to_gtv_pretty(): gtv - convert to a pretty gtv
.hash(): byte_array - returns a cryptographic hash of the value (same as .to_gtv().hash())
```

### Examples

```
val g = [1, 2, 3].to_gtv();
val l = list<integer>.from_gtv(g); // Returns [1, 2, 3]
print(g.hash());
```

---

## Collection types

Reli supports the following collection types:

- `list<T>` - an ordered list
- `set<T>` - an unordered set, contains no duplicates
- `map<K, V>` - a key-value map

Collection types are mutable, elements can be added or removed dynamically. Only a non-mutable type can be used as a map key or a set element.

### Mutable collection types

- Collection types (`list`, `set`, `map`) - always.
- Nullable type - only if the underlying type is mutable.
- Struct type - if the struct has a mutable field, or a field of a mutable type.
- Tuple - if a type of an element is mutable.

### Creating collections

```
// list
val l1 = [ 1, 2, 3, 4, 5 ];
val l2 = list<integer>();
```

(continues on next page)

(continued from previous page)

```
// set
val s = set<integer>();

// map
val m1 = [ 'Bob' : 123, 'Alice' : 456 ];
val m2 = map<text, integer>();
```

## list<T>

A list is an ordered collection type that accepts duplication of elements.

```
var messages = message @* { } ( @sort timestamp = .timestamp );
messages.add(new_message);
```

### Constructors

list<T>() - a new empty list

list<T>(list<T>) - a copy of the given list (list of subtype is accepted as well)

list<T>(set<T>) - a copy of the given set (set of subtype is accepted)

### Methods

.add(T): boolean - adds an element to the end, always returns true

.add(pos: integer, T): boolean - inserts an element at a position, always returns true

.add\_all(list<T>): boolean

.add\_all(set<T>): boolean

.add\_all(pos: integer, list<T>): boolean

.add\_all(pos: integer, set<T>): boolean

.clear()

.contains(T): boolean

.contains\_all(list<T>): boolean

.contains\_all(set<T>): boolean

.empty(): boolean

.index\_of(T): integer - returns -1 if element is not found

.remove(T): boolean - removes the first occurrence of the value, return true if found

.remove\_all(list<T>): boolean

.remove\_all(set<T>): boolean

.remove\_at(pos: integer): T - removes an element at a given position

.size(): integer

.\_sort() - sorts this list, returns nothing (name is \_sort, because sort is a keyword in Rell)

.sorted(): list<T> - returns a sorted copy of this list

.to\_text(): text - returns e. g. '[1, 2, 3, 4, 5]'

`.sub(start: integer[, end: integer]): list<T>` - returns a sub-list (start-inclusive, end-exclusive)

### Special operators

- `[]` - element access (read/modify)
- `in` - returns `true` if the value is in the list

## set<T>

Unordered collection type. Does *not* accept duplication.

```
var my_classmates = set<user>();  
my_classmates.add(alice); // return true  
my_classmates.add(alice); // return false
```

### Constructors

`set<T>()` - a new empty set

`set<T>(set<T>)` - a copy of the given set (set of subtype is accepted as well)

`set<T>(list<T>)` - a copy of the given list (with duplicates removed)

### Methods

`.add(T): boolean` - if the element is not in the set, adds it and returns `true`

`.add_all(list<T>): boolean` - adds all elements, returns `true` if at least one added

`.add_all(set<T>): boolean` - adds all elements, returns `true` if at least one added

`.clear()`

`.contains(T): boolean`

`.contains_all(list<T>): boolean`

`.contains_all(set<T>): boolean`

`.empty(): boolean`

`.remove(T): boolean` - removes the element, returns `true` if found

`.remove_all(list<T>): boolean` - returns `true` if at least one removed

`.remove_all(set<T>): boolean` - returns `true` if at least one removed

`.size(): integer`

`.sorted(): list<T>` - returns a sorted copy of this set (as a list)

`.to_text(): text` - returns e. g. `'[1, 2, 3, 4, 5]'`

### Special operators

- `in` - returns `true` if the value is in the set

## map<K,V>

A key/value pair collection type.

```
var dictionary = map<text, text>();
dictionary["Mordor"] = "A place where one does not simply walk into";
```

### Constructors

`map<K, V>()` - a new empty map

`map<K, V>(map<K, V>)` - a copy of the given map (map of subtypes is accepted as well)

### Methods

`.clear()`

`.contains(K) : boolean`

`.empty() : boolean`

`.get(K) : V` - get value by key (same as `[]`)

`.put(K, V)` - adds/replaces a key-value pair

`.keys() : set<K>` - returns a copy of keys

`.put_all(map<K, V>)` - adds/replaces all key-value pairs from the given map

`.remove(K) : V` - removes a key-value pair (fails if the key is not in the map)

`.size() : integer`

`.to_text() : text` - returns e. g. `'{x=123, y=456}'`

`.values() : list<V>` - returns a copy of values

### Special operators

- `[]` - get/set value by key
- `in` - returns `true` if a key is in the map

## Virtual types

A reduced data structure with Merkle tree. Type `virtual<T>` supports following types `T`:

- `list<*>`
- `set<*>`
- `map<text, *>`
- `struct`
- `tuple`

### T elements constraints

Additionally, types of all internal elements of `T` must satisfy following constraints:

- must be Gtv-compatible
- for a `map` type, the key type must be `text` (i. e. `map<text, *>`)

### Virtual types operations

- member access: `[]` for `list` and `map`, `.name` for `struct` and `tuple`

- `.to_full()`: `T` - converts the virtual value to the original value, if the value is full (all internal elements are present), otherwise throws an exception
- `.hash()`: `byte_array` - returns the hash of the value, which is the same as the hash of the original value.
- `virtual<T>.from_gtv(gtv)`: `virtual<T>` - decodes a virtual value from a Gtv.

#### Features of `virtual<T>`

- it is immutable
- reading a member of type `list<*>`, `map<*, *>`, `struct` or `tuple` returns a value of the corresponding virtual type, not of the actual member type
- cannot be converted to Gtv, so cannot be used as a return type of a query

#### Example

```
struct rec { t: text; s: integer; }

operation op(recs: virtual<list<rec>>) {
  for (rec in recs) {
    val full = rec.to_full();
    ↪value is not full
    print(full.t);
  }
}
```

#### `virtual<list<T>>`

`virtual<list<T>>.from_gtv(gtv)`: `virtual<list<T>>` - decodes a Gtv

`.empty()`: `boolean`

`.get(integer)`: `virtual<T>` - returns an element, same as `[]`

`.hash()`: `byte_array`

`.size()`: `integer`

`.to_full()`: `list<T>` - converts to the original value, fails if the value is not full

`.to_text()`: `text` - returns a text representation

#### Special operators

- `[]` - element read, returns `virtual<T>` (or just `T` for simple types)
- `in` - returns `true` if the given integer index is present in the virtual list

#### `virtual<set<T>>`

`virtual<set<T>>.from_gtv(gtv)`: `virtual<set<T>>` - decodes a Gtv

`.empty()`: `boolean`

`.hash()`: `byte_array`

`.size()`: `integer`

`.to_full()`: `set<T>` - converts to the original value, fails if the value is not full

`.to_text()`: `text` - returns a text representation

### Special operators

- `in` - returns `true` if the given value is present in the virtual set; the type of the operand is `virtual<T>>` (or just `T` for simple types)

### `virtual<map<K,V>>`

```
virtual<map<K,V>>.from_gtv(gtv) : virtual<map<K,V>> - decodes a Gtv
.contains(K) : boolean - same as operator in
.empty() : boolean
.get(K) : virtual<V> - same as operator []
.hash() : byte_array
.keys() : set<K> - returns a copy of keys
.size() : integer
.to_full() : map<K,V> - converts to the original value, fails if the value is not full
.to_text() : text - returns a text representation
.values() : list<virtual<V>> - returns a copy of values (if V is a simple type, returns list<V>)
```

### Special operators

- `[]` - get value by key, fails if not found, returns `virtual<V>` (or just `V` for simple types)
- `in` - returns `true` if a key is in the map

### `virtual<struct>`

```
virtual<R>.from_gtv(gtv) : R - decodes a Gtv
.hash() : byte_array
.to_full() : R - converts to the original value, fails if the value is not full
```

## Subtypes

If type `B` is a subtype of type `A`, a value of type `B` can be assigned to a variable of type `A` (or passed as a parameter of type `A`).

- `T` is a subtype of `T?`.
- `null` is a subtype of `T?`.
- `(T,P)` is a subtype of `(T?,P?)`, `(T?,P)` and `(T,P?)`.

## 2.5.2 Module definitions

### Table of Contents

- *Module definitions*
  - *Entity*
    - \* *Keys and Indices*
    - \* *Entity annotations*
    - \* *Changing existing entities*
  - *Object*
  - *Struct*
    - \* *struct<mutable T>*
    - \* *struct<operation>*
  - *Enum*
  - *Query*
  - *Operation*
  - *Function*
    - \* *Default parameter values*
    - \* *Named function arguments*
    - \* *Using a function as a value*
    - \* *Partial function application*
    - \* *Extendable functions*
  - *Namespace*
  - *External*
    - \* *External modules*
    - \* *Transactions and blocks*
  - *Mount names*

## Entity

Values (instances) of an entity in Rel are stored in a database, not in memory. They have to be created and deleted explicitly using Rel `create` and `delete` expressions. An in-memory equivalent of an entity in Rel is a struct.

A variable of an entity type holds an ID (primary key) of the corresponding database record, but not its attribute values.

```
entity company {
  name: text;
  address: text;
}

entity user {
  first_name: text;
  last_name: text;
  year_of_birth: integer;
```

(continues on next page)



(continued from previous page)

```
mutable salary: integer;
}
```

If attribute type is not specified, it will be the same as attribute name:

```
entity user {
  name;           // built-in type "name"
  company;        // user-defined type "company" (error if no such type)
}
```

Attributes may have default values:

```
entity user {
  home_city: text = 'New York';
}
```

An ID (database primary key) of an entity value can be accessed via the `rowid` implicit attribute (of type `rowid`):

```
val u = user @ { .name == 'Bob' };
print(u.rowid);

val alice_id = user @ { .name == 'Alice' } ( .rowid );
print(alice_id);
```

## Keys and Indices

Entities can have key and index clauses:

```
entity user {
  name: text;
  address: text;
  key name;
  index address;
}
```

Keys and indices may have multiple attributes:

```
entity user {
  first_name: text;
  last_name: text;
  key first_name, last_name;
}
```

Mutability can be specified within a key or index clause. Here one can also set a default value:

```
entity address{
  index mutable city: text = 'Rome';
}
```

Attribute definitions can be combined with key or index clauses, but such definition has restrictions (e. g. cannot specify mutable):

```
entity user {
  key first_name: text, last_name: text;
```

(continues on next page)

(continued from previous page)

```
index address: text;
}
```

## Entity annotations

```
@log entity user {
    name: text;
}
```

The `@log` annotation has following effects:

- Special attribute `transaction` of type `transaction` is added to the entity.
- When an entity value is created, `transaction` is set to the result of `op_context.transaction` (current transaction).
- Entity cannot have mutable attributes.
- Values cannot be deleted.

## Changing existing entities

When starting a Rel app, database structure update is performed: tables for new entities and objects are created and tables for existing ones are altered. There are limitations on changes that can be made in existing entities and objects.

What's allowed:

- Adding an attribute with a default value (a column is added to the table and initialized with the default value).
- Adding an attribute without a default value - only if there are no records in the table.
- Removing an attribute (database column is not dropped, and the attribute can be read later).

What's not allowed:

- Any changes in keys/indices, including adding a new key/index attribute, making an existing attribute into key/index, removing an attribute from an index, etc.
- Changing attribute's type.
- Adding/removing the `@log` annotation.

## Object

Object is similar to entity, but there can be only one instance of an object:

```
object event_stats {
    mutable event_count: integer = 0;
    mutable last_event: text = 'n/a';
}
```

### Reading object attributes

```
query get_event_count () = event_stats.event_count;
```

### Modifying an object

```
operation process_event(event: text) {
    update event_stats ( event_count += 1, last_event = event );
}
```

### Features of objects

- Like entities, objects are stored in a database.
- Objects are initialized automatically during blockchain initialization.
- Cannot create or delete an object from code.
- Attributes of an object must have default values.

### Struct

Struct is similar to entity, but its values exist in memory, not in a database.

```
struct user {
    name: text;
    address: text;
    mutable balance: integer = 0;
}
```

### Features of structs

- Attributes are immutable by default, and only mutable when declared with `mutable` keyword.
- Attributes can have
- An attribute may have a default value, which is used if the attribute is not specified during construction.
- Structs are deleted from memory implicitly by a garbage collector.

Creating struct values

```
val u = user(name = 'Bob', address = 'New York');
```

A struct-copy of an entity or object can also be created with `.to_struct()`

```
entity user{
    name;
    address: text;
}

val u = user @ { .name == 'Bob' };
val s = u.to_struct(); // returns struct <user>
```

Instead of specifying individual attributes in a create expression, we can pass a `struct<entity>`.

```
create user(s); // s is struct<user>
```

Same rules as for the `create` expression apply: no need to specify attribute name if it can be resolved implicitly by name or type:

```
val name = 'Bob';
val address = 'New York';
val u = user(name, address);
val u2 = user(address, name); // Order does not matter - same struct value is created.
```

Struct attributes can be accessed using operator `.`:

```
print(u.name, u.address);
```

Safe-access operator `?.` can be used to read or modify attributes of a nullable struct:

```
val u: user? = find_user('Bob');
u?.balance += 100;           // no-op if 'u' is null
```

## struct<mutable T>

struct<mutable T> has the same attributes as struct, but all attributes are mutable

To convert an entity or object to a mutable struct, one uses `.to_mutable_struct()`

```
val u = user @ { .name == 'Bob' };
val s = u.to_mutable_struct();    // will return a struct<mutable user>
```

To convert between struct<T> and struct<mutable T>, one instead uses `.to_mutable()` and `.to_immutable()`

```
val s = u.to_struct();
val mut = s.to_mutable();
val imm = mut.to_immutable();
```

## struct<operation>

The type struct<operation> defines a struct which has same attributes as a given operations parameters:

```
operation add_user(name: text, rating: integer) {
    // ...
}

query can_add_user(user: struct<add_user>) {
    if (user.name == '') return false;
    if (user.rating < 0) return false;
    return true;
}
```

## Enum

### Enum declaration

```
enum currency {
    USD,
    EUR,
    GBP
}
```

Values are stored in a database as integers. Each constant has a numeric value equal to its position in the enum (the first value is 0).

### Usage

```
var c: currency;
c = currency.USD;
```

Enum-specific functions and properties:

```
val cs: list<currency> = currency.values() // Returns all values (in the order in_
↳which they are declared)

val eur = currency.value('EUR') // Finds enum value by name
val gbp = currency.value(2) // Finds enum value by index

val usd_str: text = currency.USD.name // Returns 'USD'
val usd_value: integer = currency.USD.value // Returns 0.
```

## Query

- Cannot modify the data in the database (compile-time check).
- Must return a value.
- If return type is not explicitly specified, it is implicitly deduced.
- Parameter types and return type must be Gtv-compatible.

### Short form

```
query q(x: integer): integer = x * x;
```

### Full form

```
query q(x: integer): integer {
    return x * x;
}
```

## Operation

- Can modify the data in the database.
- Does not return a value.
- Parameter types must be Gtv-compatible.

```
operation create_user(name: text) {
    create user(name = name);
}
```

## Function

- Can return nothing or a value.
- Can modify the data in the database when called from an operation (run-time check).
- Can be called from queries, operations or functions.
- If return type is not specified explicitly, it is `unit` (no return value).

### Short form

```
function f(x: integer): integer = x * x;
```

### Full form

```
function f(x: integer): integer {  
    return x * x;  
}
```

### Return type not specified

When return type is not specified, it is considered unit:

```
function f(x: integer) {  
    print(x);  
}
```

### Default parameter values

Parameters of functions can have default values. If no parameters are specified in the function call, the default parameters will be used.

```
function f(user: text = 'Bob', score: integer = 123) {...}  
...  
f(); // means f('Bob', 123)  
f('Alice'); // means f('Alice', 123)  
f(score=456); // means f('Bob', 456)
```

### Named function arguments

One could also specify function arguments by names.

```
function f(x: integer, y: text) {}  
...  
f(x = 123, y = 'Hello');
```

### Using a function as a value

If one would want to pass a function to another function, the function to be passed can be used as a value by using the following syntax:

```
() -> boolean  
(integer) -> text  
(byte_array, decimal) -> integer
```

Within the parentheses, the function input type(s) of the passed function is specified. After the arrow follows the function's return type.

An example could look like this:

```
function filter(values: list<integer>, predicate: (integer) -> boolean): list<integer>  
→ {  
    return values @* { predicate($) };  
}
```

## Partial function application

If one would want to create a reference to a function i. e. to obtain a value of a function, the wildcard symbol `*` could be used.

```
function f(x: integer, y: integer) = x * y;

val g = f(*);           // Type of "g" is (integer, integer) -> integer
g(123, 456);           // Invocation of f(123, 456) via "g".
```

## Extendable functions

A function can be declared as extendable by adding `@extendable` in front of the function declaration. An arbitrary number of extensions can be defined for an extendable function by expressing `@extend` in front of the function declaration.

In the example below, function `f` is a base function and functions `g` and `h` are extension functions.

```
@extendable function f(x: integer) {
    print('f', x);
}

@extend(f) function g(x: integer) {
    print('g', x);
}

@extend(f) function h(x: integer) {
    print('h', x);
}
```

When the base function is called, all its extension functions are executed, and the base function itself is executed in the end. However, Extendable functions support a limited set of return types and this behavior depends on the return type. The following behavior applies to the different return types:

- **unit**
  - all extensions are executed
  - base function is always executed in the end
- **boolean**
  - extensions are executed one by one, until some of them returns “true”
  - base function is executed if all extensions returned “false”
  - the result of the last executed function is returned to the caller
- **T?**
  - Similar to boolean. Extensions are executed until the first non-null result, which is returned to the caller
- **list<T>**
  - all extensions are executed
  - the base function is executed in the end
  - the concatenation of all lists is returned to the caller

- `map<K, V>`
  - similar to `list<T>`, the union of all maps is returned to the caller, but fails if there is a key conflict

## Namespace

Definitions can be put in a namespace:

```
namespace foo {
  entity user {
    name;
    country;
  }

  struct point {
    x: integer;
    y: integer;
  }

  enum country {
    USA,
    DE,
    FR
  }
}

query get_users_by_country(c: foo.country) = foo.user @* { .country == c };
```

Features of namespaces:

- No need to specify a full name within a namespace, i.e. can use `country` under namespace `foo` directly, not as `foo.country`.
- Names of tables for entities and objects defined in a namespace contain the full name, e. g. the table for entity `foo.user` will be named `c0.foo.user`.
- It is allowed to define namespace with same name multiple times with different inner definitions.

Anonymous namespace:

```
namespace {
  // some definitions
}
```

Can be used to apply an annotation to a set of definitions:

```
@mount('foo.bar')
namespace {
  entity user {}
  entity company {}
}
```

Short nested namespace notation:

```
namespace x.y.z {
  function f() = 123;
}
```

Is equivalent to:



```
namespace x {
  namespace y {
    namespace z {
      function f() = 123;
    }
  }
}
```

Splitting namespace between files

One can split a namespace between different files in a module like so:

```
lib/a.rell:

namespace ns { function f(): integer = 123; }
```

```
lib/b.rell:

namespace ns { function g(): integer = 456; }
```

Which later can be accessed like so:

```
main.rell:

import lib;
// ...
lib.f();
lib.g();
```

## External

The `@external` annotation is used to access entities defined in other blockchains.

```
@external('foo') namespace {
  @log entity user {}
  @log entity company {}
}

@external('foo') @log entity city {}

query get_all_users() = user @* {};
```

In this example, 'foo' is the name of an external blockchain. To be used in an `@external` annotation, a blockchain must be defined in the blockchain configuration (dependencies node).

Every blockchain has its `chain_id`, which is included in table names for entities and objects of that chain. If the blockchain 'foo' has `chain_id` = 123, the table for the entity user will be called `c123.user`.

## Features

- External entities must be annotated with the `@log` annotation. This implies that those entities cannot have mutable attributes.
- Values of external entities cannot be created or deleted.
- Only entities, namespaces and imports can be annotated with `@external`.

- When selecting values of an external entity (using at-expression), an implicit block height filter is applied, so the active blockchain can see only those blocks of the external blockchain whose height is lower than a specific value.
- Every blockchain stores the structure of its entities in meta-information tables. When a blockchain is started, the meta-information of all involved external blockchains is verified to make sure that all declared external entities exist and have declared attributes.

## External modules

A module can be annotated with the `@external` with no arguments:

```
@external module;  
  
@log entity user {}  
@log entity company {}
```

### Features

- can contain only namespaces, entities (annotated with `@log`) and imports of other external modules;
- can be imported as a regular or an external module.

Regular import: entities defined in the module `ext` belong to the current blockchain.

```
import ext;
```

External import: entities defined in the module `ext` are imported as external entities from the chain `foo`.

```
@external('foo') import ext;
```

## Transactions and blocks

To access blocks and transactions of an external blockchain, a special syntax is used:

```
@external('foo') namespace foo {  
    entity transaction;  
    entity block;  
}  
  
function get_foo_transactions(): list<foo.transaction> = foo.transaction @* {};  
function get_foo_blocks(): list<foo.block> = foo.block @* {};
```

- External and non-external transactions/blocks are distinct, incompatible types.
- When selecting external transactions or blocks, an implicit height filter is applied (like for external entities).

Entities `transaction` and `block` of an external chain can be accessed also via an external module:

```
@external('foo') import ext;  
  
function get_foo_transactions(): list<ext.transaction> = ext.transaction @* {};  
function get_foo_blocks(): list<ext.block> = ext.block @* {};
```

The entities are implicitly added to the module's namespace and can be accessed by its import alias.

## Mount names

Entities, objects, operations and queries have mount names:

- for entities and objects, those names are the SQL table names where the data is stored
- for operations and queries, a mount name is used to invoke an operation or a query from the outside

By default, a mount name is defined by a fully-qualified name of a definition:

```
namespace foo {
  namespace bar {
    entity user {}
  }
}
```

The mount name for the entity `user` is `foo.bar.user`.

### Custom mount names

To specify a custom mount name, `@mount` annotation is used:

```
@mount('foo.bar.user')
entity user {}
```

The `@mount` annotation can be specified for entities, objects, operations and queries.

### Mount for namespace

```
@mount('foo.bar')
namespace ns {
  entity user {}
}
```

The mount name of `user` is `foo.bar.user`.

### Mount for module

```
@mount('foo.bar')
module;

entity user {}
```

The mount name of `user` is `foo.bar.user`.

### Nested namespace mounts

A mount name can be relative to the context mount name. For example, when defined in a namespace

```
@mount('a.b.c')
namespace ns {
  entity user {}
}
```

entity `user` will have following mount names when annotated with `@mount`:

- `@mount('.d.user')` -> `a.b.c.d.user`
- `@mount('^..user')` -> `a.b.user`
- `@mount('^^.x.user')` -> `a.x.user`

Special character `.` appends names to the context mount name, and `^` removes the last part from the context mount name.

A mount name can end with `.`, in that case the name of the definition is appended to the mount name:

```
@mount ('foo.')
entity user {}           // mount name = "foo.user"

@mount ('foo')
entity user {}           // mount name = "foo"
```

## 2.5.3 Expressions

### Table of Contents

- *Expressions*
  - *Values*
  - *Operators*
    - \* *Special*
    - \* *Comparison*
    - \* *Arithmetical*
    - \* *Logical*
    - \* *If*
    - \* *Other*

---

## Values

Simple values:

- Null: `null` (type is `null`)
- Boolean: `true`, `false`
- Integer: `123`, `0`, `-456`
- Text: `'Hello'`, `"World"`
- Byte array: `x'1234'`, `x"ABCD"`

Text literals may have escape-sequences:

- Standard: `\r`, `\n`, `\t`, `\b`.
- Special characters: `\"`, `\'`, `\\`.
- Unicode: `\u003A`.

## Operators

### Special

- `.` - member access: `user.name, s.sub(5, 10)`
- `()` - function call: `print('Hello'), value.to_text()`
- `[]` - element access: `values[i]`

### Comparison

- `==`
- `!=`
- `===`
- `!==`
- `<`
- `>`
- `<=`
- `>=`

Operators `==` and `!=` compare values. For complex types (collections, tuples, structs) they compare member values, recursively. For entity values only object IDs are compared.

Operators `===` and `!==` compare references, not values. They can be used only on types: `tuple`, `struct`, `list`, `set`, `map`, `gtv`, `range`.

Example:

```
val x = [1, 2, 3];
val y = list(x);
print(x == y);      // true - values are equal
print(x === y);     // false - two different objects
```

### Arithmetical

- `+`
- `-`
- `*`
- `/`
- `%`
- `++`
- `--`

## Logical

- `and`
- `or`
- `not`

## If

Operator `if` is used for conditional evaluation:

```
val max = if (a >= b) a else b;  
return max;
```

## Other

- `in` - check if an element is in a range/set/map
- `not in` - check if an element is not in a range/set/map

## 2.5.4 Statements

### Table of Contents

- *Statements*
  - *Local variable declaration*
  - *Basic statements*
  - *If statement*
  - *When statement*
  - *Loop statements*

## Local variable declaration

Constants:

```
val x = 123;  
val y: text = 'Hello';
```

Variables:

```
var x: integer;  
var y = 123;  
var z: text = 'Hello';
```

## Basic statements

Assignment:

```
x = 123;
values[i] = z;
y += 15;
```

Function call:

```
print('Hello');
```

Return:

```
return;
return 123;
```

Block:

```
{
    val x = calc();
    print(x);
}
```

## If statement

```
if (x == 5) print('Hello');

if (y == 10) {
    print('Hello');
} else {
    print('Bye');
}

if (x == 0) {
    return 'Zero';
} else if (x == 1) {
    return 'One';
} else {
    return 'Many';
}
```

Can also be used as an expression:

```
function my_abs(x: integer): integer = if (x >= 0) x else -x;
```

## When statement

Similar to switch in C++ or Java, but using the syntax of when in Kotlin:

```
when(x) {
    1 -> return 'One';
    2, 3 -> return 'Few';
    else -> {
```

(continues on next page)

(continued from previous page)

```

    val res = 'Many: ' + x;
    return res;
}

```

## Features

- Can use both constants as well as arbitrary expressions.
- When using constant values, the compiler checks that all values are unique.
- When using with an enum type, values can be specified by simple name, not full name.

A form of when without an argument is equivalent to a chain of if... else if:

```

when {
  x == 1 -> return 'One';
  x >= 2 and x <= 7 -> return 'Several';
  x == 11, x == 111 -> return 'Magic number';
  some_value > 1000 -> return 'Special case';
  else -> return 'Unknown';
}

```

- Can use arbitrary boolean expressions.
- When multiple comma-separated expressions are specified, any of them triggers the block (i. e. they are combined via OR).

Both forms of when (with and without an argument) can be used as an expression:

```

return when(x) {
  1 -> 'One';
  2, 3 -> 'Few';
  else -> 'Many';
}

```

- else must always be specified, unless all possible values of the argument are specified (possible for boolean and enum types).
- Can be used in at-expression, in which case it is translated to SQL CASE WHEN... THEN expression.

## Loop statements

For:

```

for (x in range(10)) {
  print(x);
}

for (u in user @* {}) {
  print(u.name);
}

```

The expression after in may return a range or a collection (list, set, map).

Tuple unpacking can be used in a loop:

```

val l: list<(integer, text)> = get_list();
for ((n, s) in l) { ... }

```



While:

```
while (x < 10) {
    print(x);
    x = x + 1;
}
```

Break:

```
for (u in user @* {}) {
    if (u.company == 'Facebook') {
        print(u.name);
        break;
    }
}

while (x < 5) {
    if (values[x] == 3) break;
    x = x + 1;
}
```

Continue:

```
for (u in user @* {}) {
    if (u.company == 'BigCompanyCo') {
        continue;
    }
    print(u.name); // Will print every user who does not work at BigCompanyCo.
}
```

## 2.5.5 Database Operations

### Table of Contents

- *Database Operations*
  - *At-Operator*
    - \* *Cardinality*
    - \* *From-part*
    - \* *Where-part*
    - \* *What-part*
    - \* *Tail part*
    - \* *Result type*
    - \* *Nested At-Operators*
    - \* *Aggregate functions and grouping*
    - \* *Sorting*
    - \* *Field names*
  - *Create Statement*

- *Update Statement*
- *Delete Statement*

## At-Operator

By using the At-Operator, database records can be retrieved. The general syntax consists of five parts, some of which can be omitted: <from> <cardinality> { <where> } [<what>] [limit N]

### Simplest form

```
user @ { .name == 'Bob' }
```

## Cardinality

Specifies whether the expression must return one or many objects:

- `T @? {}` - returns `T?`, zero or one, fails if more than one found.
- `T @ {}` - returns `T`, exactly one, fails if zero or more than one found.
- `T @* {}` - returns `list<T>`, zero or more.
- `T @+ {}` - returns `list<T>`, one or more, fails if none found.

## From-part

The from-part is declared before `@` and specifies which entity type(s) that will be retrieved.

### Simple (one entity)

```
user @* { .name == 'Bob' }
```

### Complex (one or more entities)

```
(user, company) @* { user.name == 'Bob' and company.name == 'Microsoft' and  
user.xyz == company.xyz }
```

### Specifying entity aliases

```
(u: user) @* { u.name == 'Bob' }
```

```
(u: user, c: company) @* { u.name == 'Bob' and c.name == 'Microsoft' and  
u.xyz == c.xyz }
```

## Where-part

The where-part is declared after `@` and specifies which instances of the entity type that will be retrieved based on attributes. The records are filtered by zero or more comma-separated expressions using entity attributes, local variables or system functions.

- `user @* {}` - returns all users
- `user @ { .name == 'Bill', .company == 'Microsoft' }` - returns a specific user (all conditions must match)

- Attributes of an entity can be accessed with a dot, e. g. `.name` or with an entity name or alias, `user.name`.

Entity attributes can also be matched implicitly by name or type

```
val ms = company @ { .name == 'Microsoft' };
val name = 'Bill';
return user @ { name, ms };
```

Explanation: the first where-expression is the local variable `name`, there is an attribute called `name` in the entity `user`. The second expression is `ms`, there is no such attribute, but the type of the local variable `ms` is `company`, and there is an attribute of type `company` in `user`.

## What-part

The What-part is declared after the where-part and specifies which record attribute(s) that will be retrieved.

- `user @ { .name == 'Bob' } ( .company.name )` - returns a single value (name of the user's company)
- `user @ { .name == 'Bob' } ( .company.name, .company.address )` - returns a tuple of two values

## Sorting

Sorting tuples is done by using the `@sort` (ascending) and `@sort_desc` (descending) annotation.

- `user @* {} ( @sort .last_name, @sort .first_name )` - sort by `last_name` first, then by `first_name`
- `user @* {} ( @sort_desc .year_of_birth, @sort .last_name )` - sort by `year_of_birth` descending, then by `last_name` ascending

## Result tuple fields

Returned tuples can have named fields.

- `user @* {} ( x = .company.name, y = .company.address, z = .year_of_birth )` - returns a tuple with named fields (`x`, `y`, `z`)
- `user @* {} ( @sort x = .last_name, @sort_desc y = .year_of_birth )` - field names can be combined with sorting

When field names are not specified explicitly, they can be inferred implicitly from attribute name:

```
val u = user @ { ... } ( .first_name, .last_name, age = 2018 - .year_of_birth );
print(u.first_name, u.last_name, u.age);
```

By default, if a field name is not specified and the expression is a single name (e. g. an attribute of an entity), that name is used as a tuple field name:

```
val u = user @ { ... } ( .first_name, .last_name );
// Result is a tuple (first_name: text, last_name: text).
```

To have a tuple field without a name, use `_` as field name:

```
val u = user @ { ... } ( _ = .first_name, _ = .last_name );
// Result is a tuple (text, text).
```

To exclude a field from the result tuple, use `@omit` annotation:

```
val us = user @* {} ( .last_name, @omit .first_name ) ;
// Result is list<text>, since first_name is excluded, so there is only one_
↳expression to return
```

The possibility to exclude a field is useful, for example, when one needs to sort by some expression, but does not want to include that expression into the result tuple:

```
val sorted_users = user @* {} ( _ = .first_name, _ = .last_name, @omit @sort .date_of_
↳birth );
// Returns list<(text,text)>.
```

## Tail part

The tail part is declared after the What-part and filters the returned tuples. This is done by limiting and skipping records.

### Limiting records

- `user @* { .company == 'Microsoft' } limit 10` - Returns at most 10 objects.

The limit is applied before the cardinality check, so the following code can't fail with "more than one object" error:

- `val u: user = user @ { .company == 'Microsoft' } limit 1;` - Returns one record

### Skipping records

- `user @* {} (@sort .company) offset 10` - Skips the first 10 records in the table, can be used alongside limit to specify a subset within the found records.
- `people @* {} (@sort .age) offset 10 limit 20` - Returns the 11th youngest up to the 30th youngest person.

## Result type

The Result type depends on the cardinality, from- and what-parts.

- From- and what-parts define the type of a single record, T.
- Cardinality defines the type of the @-operator result: T?, T or list<T>.

### Examples

- `user @ { ... }` - returns user
- `user @? { ... }` - returns user?
- `user @* { ... }` - returns list<user>
- `user @+ { ... }` - returns list<user>
- `(user, company) @ { ... }` - returns a tuple (user, company)
- `(user, company) @* { ... }` - returns list<(user, company)>
- `user @ { ... } ( .name )` - returns text
- `user @ { ... } ( .first_name, .last_name )` - returns (first\_name:text, last\_name:text)
- `(user, company) @ { ... } ( user.first_name, user.last_name, company )` - returns (text, text, company)

## Nested At-Operators

A nested at-operator can be used in any expression inside of another at-operator:

```
user @* { .company == company @ { .name == 'Microsoft' } } ( ... )
```

This is equivalent to:

```
val c = company @ { .name == 'Microsoft' };
user @* { .company == c } ( ... )
```

## Aggregate functions and grouping

Equivalents to SQL statements GROUP BY, MIN, MAX and SUM can be expressed by using @group, @min, @max and @sum.

- To calculate an aggregated value (min/max/sum) use @min, @max or @sum.
- Grouping by an attribute (or an expression) is done by annotating it with @group
- To calculate count, use @sum 1.

### Example entity

```
entity city {
    name;
    country: text;
    population: integer;
}
```

### Examples

To calculate the number of cities in every country and grouping it by country, one can write:

```
city @*{} ( @group .country, @sum 1 )
```

The result is a list of tuples (text, integer) - (country name, number of cities)

Calculating the total population of all cities in each country can be expressed as:

```
city @*{} ( @group .country, @sum 1, @sum .population )
```

Note that more than one expression can be annotated with @group in order to group by multiple values.

## Sorting

Sorting can be done by using @sort with or without the @omit statement.

```
city @*{} ( @group .country, @omit @sort_desc @sum 1 )
```

In the example above, the countries are sorted by the number of cities in a descending order. Note that the @omit statement is included, hence the number of cities is not displayed in the result.

## Field names

Tuple field names can be specified after annotations:

```
city @*{ } ( @group .country, @sum city_count = 1, @sum total_population = .population_
↪ )
```

## Create Statement

Must specify all attributes that don't have default values.

```
create user(name = 'Bob', company = company @ { .name == 'Amazon' }));
```

No need to specify attribute name if it can be matched by name or type:

```
val name = 'Bob';
create user(name, company @ { company.name == 'Amazon' }));
```

Can use the created object:

```
val new_company = create company(name = 'Amazon');
val new_user = create user(name = 'Bob', new_company);
print('Created new user:', new_user);
```

## Update Statement

Operators @, @?, @\*, @+ are used to specify cardinality, like for the at-operator. If the number of updated records does not match the cardinality, a run-time error occurs.

```
update user @ { .name == 'Bob' } ( company = 'Microsoft' );           // exactly one
update user @? { .name == 'Bob' } ( deleted = true );                 // zero or one
update user @* { .company.name == 'Bad Company' } ( salary -= 1000 ); // any number
```

Can change only mutable attributes.

Entity attributes can be matched implicitly by name or type:

```
val company = 'Microsoft';
update user @ { .name == 'Bob' } ( company );
```

Using multiple entities with aliases. The first entity is the one being updated. Other entities can be used in the where-part:

```
update (u: user, c: company) @ { u.xyz == c.xyz, u.name == 'Bob', c.name == 'Google' }
↪ ( city = 'Seattle' );
```

Can specify an arbitrary expression returning a entity, a nullable entity or a collection of entities:

```
val u = user @? { .name == 'Bob' };
update u ( salary += 5000 );
```

A single attribute of can be modified using a regular assignment syntax:

```
val u = user @ { .name == 'Bob' };
u.salary += 5000;
```

## Delete Statement

Operators @, @?, @\*, @+ are used to specify cardinality, like for the at-operator. If the number of deleted records does not match the cardinality, a run-time error occurs.

```
delete user @ { .name == 'Bob' };           // exactly one
delete user @? { .name == 'Bob' };          // zero or one
delete user @* { .company.name == 'Bad Company' }; // any number
```

Using multiple entities. Similar to update, only the object(s) of the first entity will be deleted:

```
delete (u: user, c: company) @ { u.xyz == c.xyz, u.name == 'Bob', c.name == 'Google' }
↪;
```

Can specify an arbitrary expression returning an entity, a nullable entity or a collection of entities:

```
val u = user @? { .name == 'Bob' };
delete u;
```

## 2.5.6 System Library (Globals)

### Table of Contents

- *System Library (Globals)*
  - *Entities*
  - *Namespaces*
    - \* *chain\_context*
    - \* *op\_context*
    - \* *crypto*
  - *Global Functions*
    - \* *Example of a like function*
  - *Require function*

## Entities

```
entity block {
  block_height: integer;
  block_rid: byte_array;
```

(continues on next page)

(continued from previous page)

```
    timestamp;
}

entity transaction {
    tx_rid: byte_array;
    tx_hash: byte_array;
    tx_data: byte_array;
    block;
}
```

It is not possible to create, modify or delete values of these entities in code.

## Namespaces

### chain\_context

`chain_context.args`: `module_args` - module arguments specified in `run.xml`. The type is `module_args`, which must be a user-defined struct. If no `module_args` struct is defined in the module, the `args` field cannot be accessed.

Example of `module_args`:

```
struct module_args {
    name: text;
    age: integer;
}
```

Corresponding module configuration:

```
<run wipe-db="true">
  <chains>
    <chain name="module-args-example" iid="0">
      <config height="0">
        <app module="example">
          <arg key="name"><string>Alice</string></arg>
          <arg key="age"><integer>46</integer></arg>
        </app>
      </config>
    </chain>
  </chains>
</run>
```

Code that reads `module_args`:

```
function f() {
    print(chain_context.args.name);
    print(chain_context.args.age);
}
```

Every module can have its own `module_args`. Reading `chain_context.args` returns the args for the current module, and the type of `chain_context.args` is different for different modules: it is the `module_args` struct defined in that module.



`chain_context.blockchain_rid:` `byte_array` - blockchain RID

`chain_context.raw_config:` `gtv` - blockchain configuration object, e. g.  
`{"gtx":{"rell":{"mainFile":"main.rell"}}`

## op\_context

System namespace `op_context` can be used only in an operation or a function called from an operation, but not in a query

- `op_context.block_height:` `integer` - the height of the block currently being built, equivalent to `op_context.transaction.block.block_height`
- `op_context.last_block_time:` `integer` - the timestamp of the last block, in milliseconds (like `System.currentTimeMillis()` in Java). Returns `-1` if there is no last block (the block currently being built is the first block)
- `op_context.op_index:` `integer` - Index of the operation being executed in the transaction (0 == first operation)
- `op_context.get_signers():` `list<byte_array>` - Returns pubkeys of the signers of the current transaction
- `op_context.is_signer(pubkey: byte_array):` `boolean` - Checks if the pubkey is one of the signers of the current transaction
- `op_context.get_all_operations():` `list<gtx_operation>` - Returns all operations of the current transaction.
- `op_context.transaction:` `transaction` - the transaction currently being built.

## crypto

Namespace used for cryptographic functions.

`crypto.keccak256(byte_array):` `byte_array` - cryptographic hash functions  
`crypto.sha256(byte_array):` `byte_array`

---

## Global Functions

- `abs(integer):` `integer` - absolute value of an integer
- `abs(decimal):` `decimal` - absolute value of a decimal
- `empty(T?):` `boolean` - returns `true` if the argument is `null` or an empty collection and `false` otherwise; for nullable collections checks both conditions
- `empty(list<T>):` `boolean`
- `empty(set<T>):` `boolean`
- `empty(map<K, V>):` `boolean`

- `exists(T?): boolean` - opposite to `empty()`
- `exists(list<T>): boolean`
- `exists(set<T>): boolean`
- `exists(map<K, V>): boolean`
- `log(...)` - print a message to the log (same usage as `print`)
- `max(integer, integer): integer` - maximum of two integer values
- `max(decimal, decimal): decimal` - maximum of two decimal value
- `min(integer, integer): integer` - minimum of two integer values
- `min(decimal, decimal): decimal` - minimum of two decimal values
- `text.like(pattern): boolean` - simple pattern matching function, equivalent to the SQL LIKE clause. Special character “`_`” matches any single character and “`%`” matches any string of zero or more characters.

### Example of a like function

- `print(name.like(% von %))` - returns all names that have a von inside
  - `user @* {name.like(Vi_tor)}` - returns all users that have one character between Vi and tor (e.g Victor or Viktor)
  - `print(...)` - print a message to STDOUT:
  - `print()` - prints an empty line
  - `print('Hello', 123)` - prints "Hello 123"
  - `verify_signature(message: bytearray, pubkey: pubkey, signature: bytearray): boolean` - returns true if the given signature is a result of signing the message with a private key corresponding to the given public key.
- 

## Require function

### Checking a boolean condition

- `require(boolean[, text])` - throws an exception if the argument is false

### Checking for null

- `require(T?[, text]): T` - throws an exception if the argument is null, otherwise returns the argument
- `require_not_empty(T?[, text]): T` - same as the previous one

### Checking for an empty collection

- `require_not_empty(list<T>[, text]): list<T>` - throws an exception if the argument is an empty list, otherwise returns the list
- `require_not_empty(set<T>[, text]): set<T>` - throws an exception if the argument is an empty set, otherwise returns the set
- `require_not_empty(map<K,V>[, text]): map<K,V>` - throws an exception if the argument is an empty map, otherwise returns the map

When passing a nullable collection to `require_not_empty`, it throws an exception if the argument is either `null` or an empty collection.

### Examples

```
val x: integer? = calculate();
val y = require(x, "x is null"); // type of "y" is "integer", not "integer?"

val p: list<integer> = get_list();
require_not_empty(p, "List is empty");

val q: list<integer>? = try_to_get_list();
require(q);           // fails if q is null
require_not_empty(q); // fails if q is null or an empty list
```

## 2.5.7 Miscellaneous

### Comments

Single-line comment:

```
print("Hello"); // Some comment
```

Multiline comment:

```
print("Hello"/*, "World"*/);
/*
print("Bye");
*/
```

## 2.6 Advanced Topics

### 2.6.1 Modules in an Application

Rell application consists of modules. A module is either a single `.rell` file or a directory with one or multiple `.rell` files.

A single-file Rell module must have a module header:

```
module;

// entities, operations, queries, functions and other definitions
```

If a `.rell` file has no module header, it is a part of a directory-module. All such `.rell` files in a directory belong to the same directory-module. An exception is a file called `module.rell`: it always belongs to a directory-module, even if it has a module header. It is not mandatory for a directory-module to have a `module.rell`.

Every file of a directory-module sees definitions of all other files of the module. A file-module file sees only its own definitions.

Example of a Rell source directory tree:

```
.
└─ app
    └─ multi
        ├── functions.rell
        ├── module.rell
        ├── operations.rell
        └── queries.rell
    └─ single.rell
```

**app/multi/functions.rell:**

```
function g(): integer = 456;
```

**app/multi/module.rell:**

```
module;
enum state { OPEN, CLOSED }
```

**app/single.rell:**

```
module;
function f(): integer = 123;
```

Every module has a name defined by its source directory path. The sample source directory tree given above defines two modules:

- `app.multi` - a directory-module in the directory `app/multi` (consisting of 4 files)
- `app.single` - a file-module in the file `app/single.rell`

There may be a root module - a directory-module which consists of `.rell` files located in the root of the source directory. Root module has an empty name. Web IDE uses the root module as the default main module of a Rel application.

## Import

To access module's definitions, the module has to be imported:

```
import app.single;

function test() {
    single.f();           // Calling the function "f" defined in the module "app.single"
    ↪ "
}
```

When importing a module, it is added to the current namespace with some alias. By default, the alias is the last part of the module name, i. e. `single` for the module `app.single` or `multi` for `app.multi`. The definitions of the module can be accessed via the alias.

A custom alias can be specified:

```
import alias: app.multi;

function test() {
    alias.g();
}
```

It is possible to specify a relative name of a module when importing. In that case, the name of the imported module is derived from the name of the current module. For example, if the current module is `a.b.c`,

- `import .d; imports a.b.c.d`
- `import alias: ^; imports a.b`
- `import alias: ^^; imports a`
- `import ^.e; imports a.b.e`

## Wildcard imports

Importing all definitions of a module:

```
import foo.*;
```

All definitions are added directly to the importing namespace.

It is possible to import definitions of a specific namespace defined within a module:

```
import foo.{ns.*};
```

An import alias, if specified, creates a nested namespace and adds imported definitions there:

```
import sub: foo.{ns.*};
```

Definitions from the namespace “ns” of module “foo” will in this example be added to a new namespace “sub”.

## Import of specific definitions

To import a specific definition (or a set of definitions) from a module, specify their names in braces:

```
import foo.{f};
import foo.{g, h};
```

The definitions “f”, “g” and “h” are added to the importing namespace like if they were defined there.

If an import alias is specified, a nested namespace is created:

```
import ns: foo.{f, g};
```

This creates a namespace “ns” containing definitions “f” and “g”.

One can specify an alias for individual definitions in braces:

```
import foo.{a: f, b: g};
```

Imported definitions will in this example be added to the namespace under names “a” and “b”.

## Run-time

At run-time, not all modules defined in a source directory tree are active. There is a main module which is specified when starting a Rel application. Only the main module and all modules imported by it (directly or indirectly) are active.

When a module is active, its operations and queries can be invoked, and tables for its entities and objects are added to the database on initialization.

## 2.6.2 Chromia Vault

### Short Overview

Chromia Vault is a wallet in nature that supports asset transfers within the Chromia ecosystem. It can be used to transfer any kind of *FT3* assets (Chromia equivalent of Ethereum ERC-20 and ERC-721 protocols).

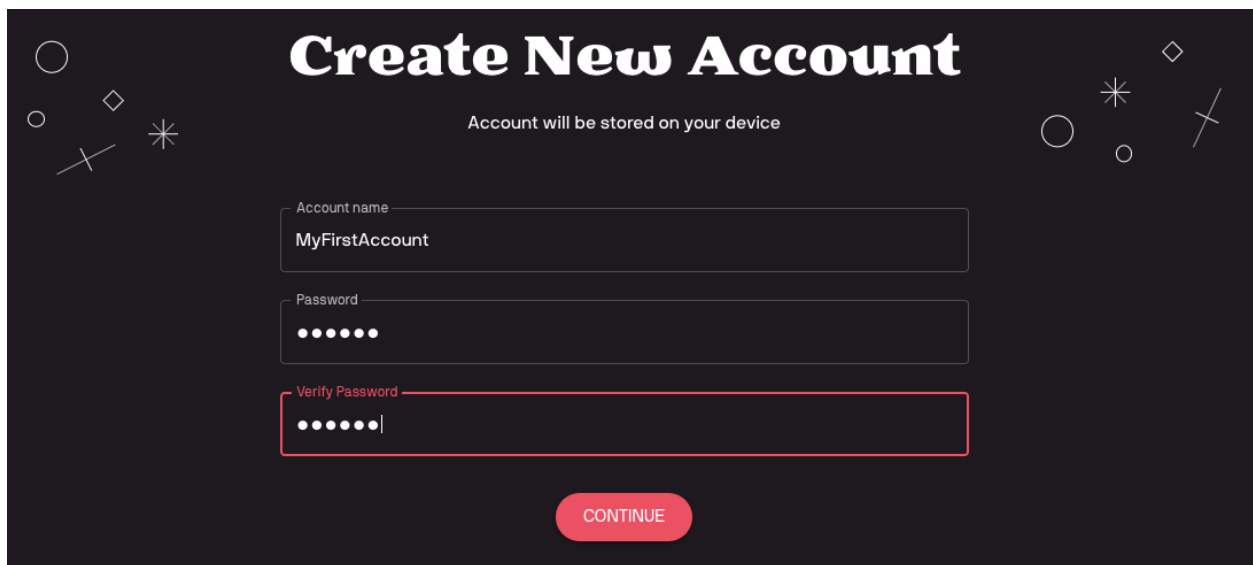
However, besides just transfers, Vault has additional features like dapp account linking and browsing Chromia dapps. Dapp account linking feature allows you to Single Sign-On (SSO) into your dapp account using the Vault (same way Google or Facebook login can be used to login into different websites), and to control your dapp account assets directly from the Vault.

This section describe how an end user can perform such actions on the [Chromia Vault webapp](#).

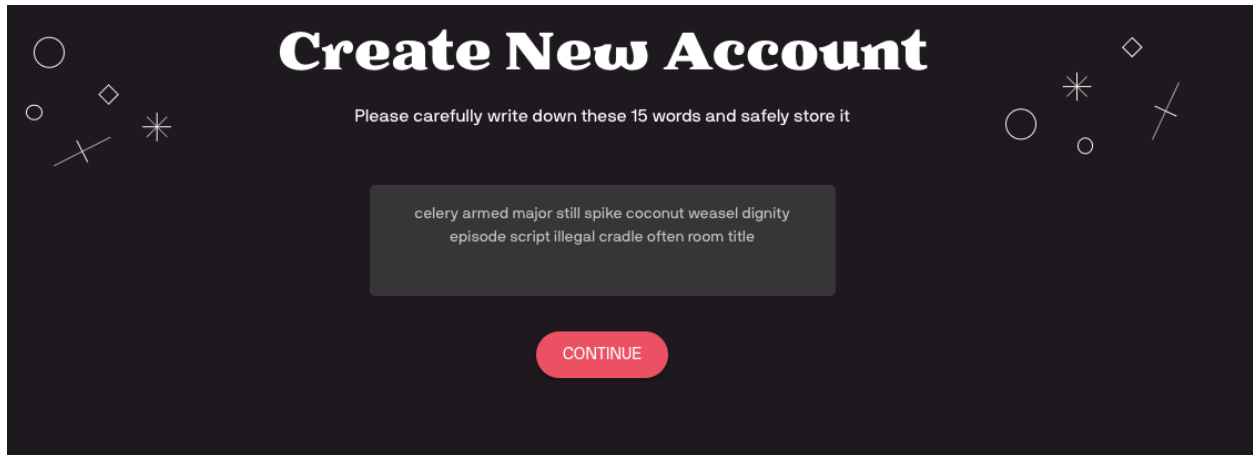
### Accessing the Vault

#### Creating new account

Account creation is a 3-steps process.

The image shows a dark-themed web form titled "Create New Account". Below the title, a subtitle reads "Account will be stored on your device". The form contains three input fields: "Account name" with the text "MyFirstAccount", "Password" with six dots, and "Verify Password" with six dots and a cursor. A red border highlights the "Verify Password" field. At the bottom of the form is a red "CONTINUE" button. The background of the form has decorative white geometric shapes like circles, diamonds, and crosses.

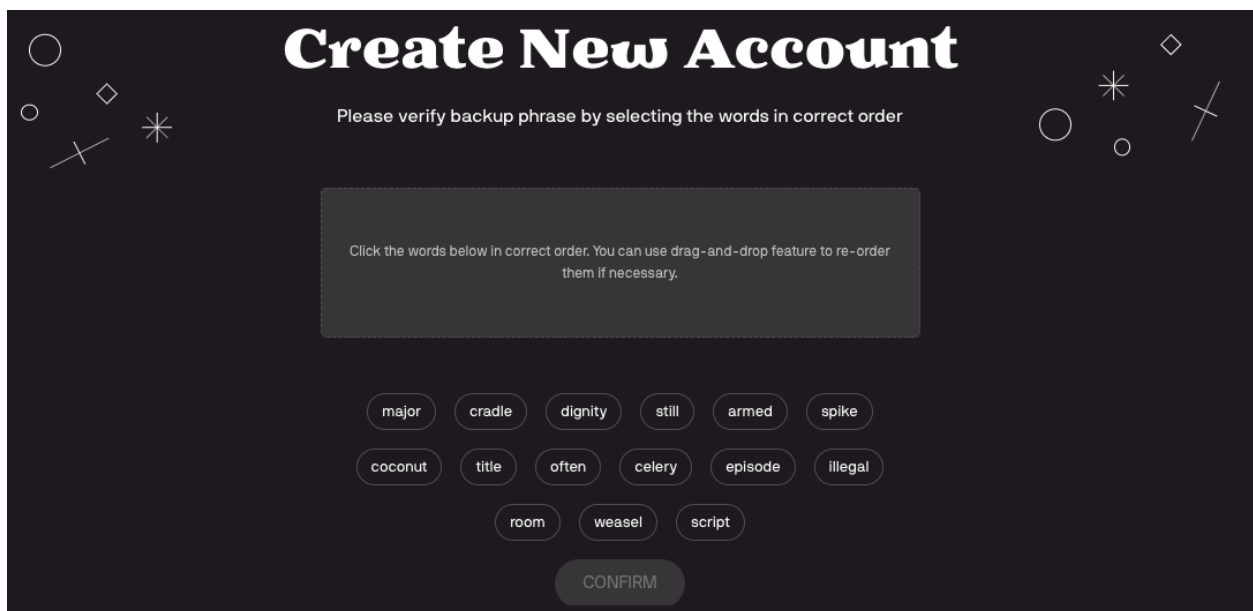
1. First, user will have to choose the account name along with a password for it. The name and password are not public, and will only be used for accessing the account later on.



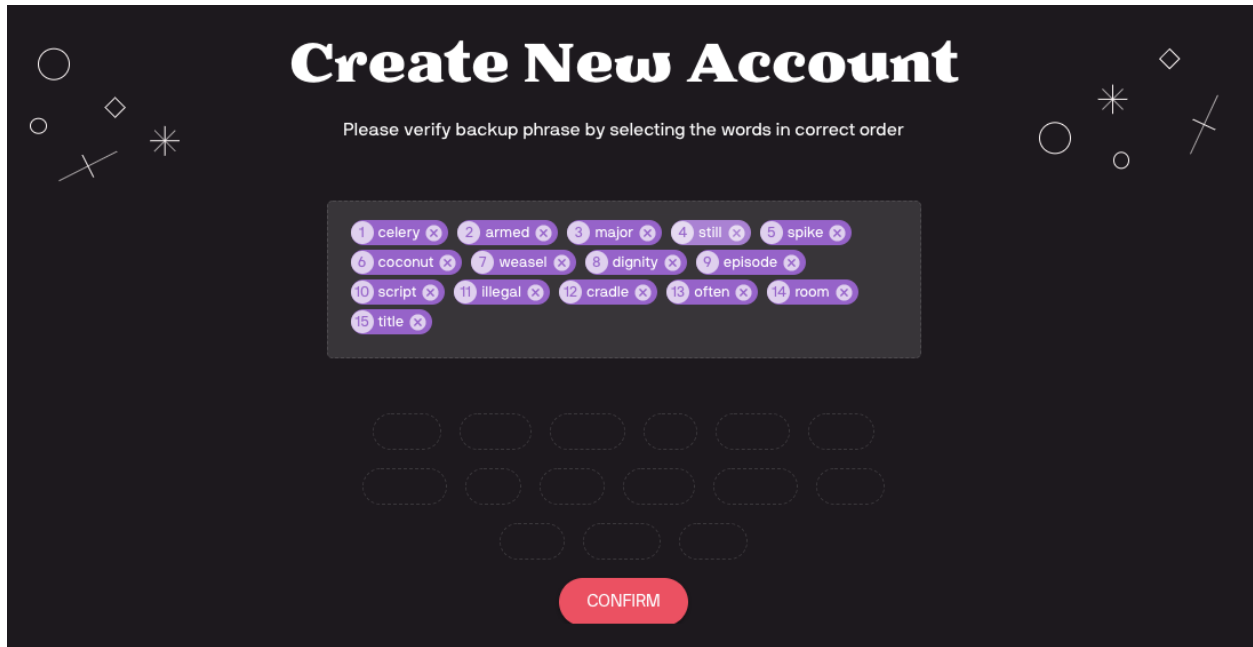
2. After account name and password fields are filled, by clicking “Continue” user will be taken to the new screen showing 15 words (the **mnemonic**).

It is highly advised to print out or write down these words and store them in a safe place.

**Important:** Knowing these 15 words in correct order is the only way to retrieve the account if the password has been lost or the browser history is cleared for any reason.



3. After user has safely stored the words, clicking “Continue” will take user to a screen where they will have to click (or drag&drop) the words in the correct order and thus “confirm” that they has stored the words somewhere.

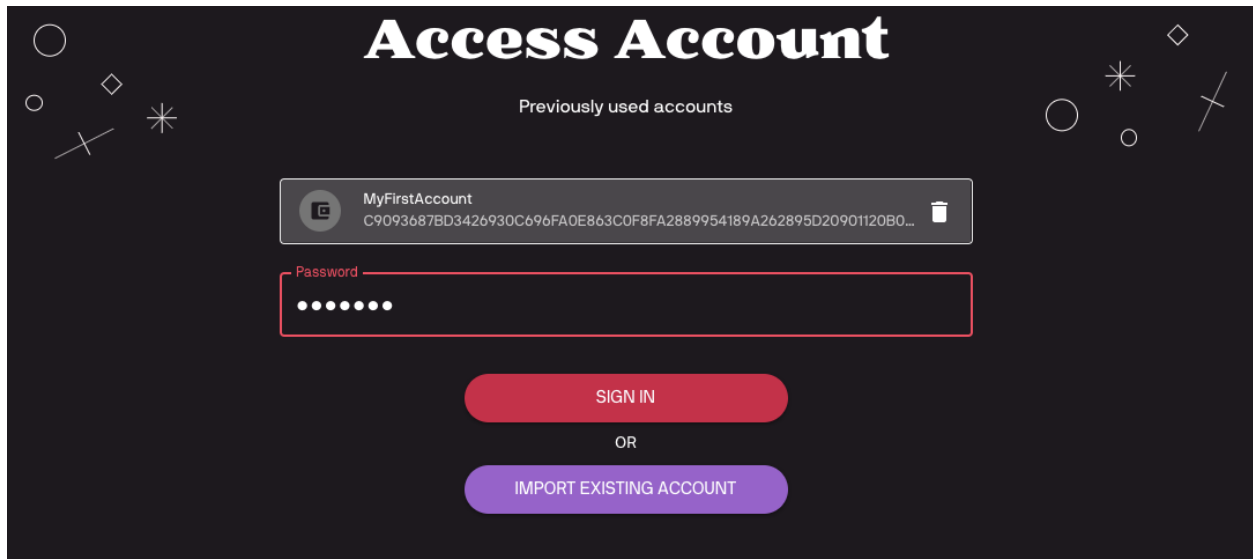


When all words are laid down in the correct order will the “Confirm” button be enabled.

By clicking “Confirm”, the account will be stored to local browser storage and user will be taken to the Dashboard screen.

### Accessing the account (Login)

All created accounts will be stored to local browser storage (on the device). Accounts are unlocked by a password that was chosen while creating an account.



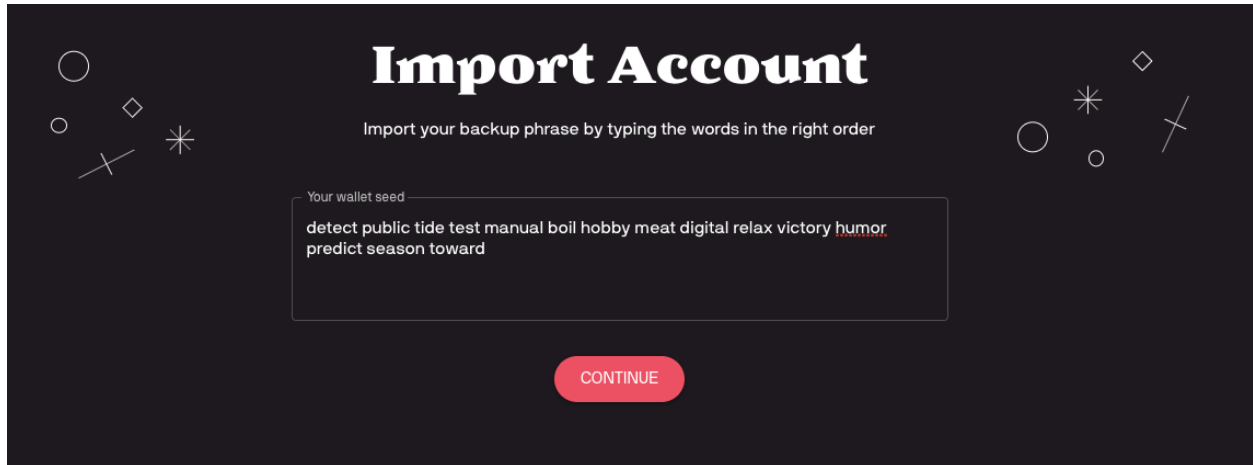
If the user wants to access an existing account on a device, they can use the “Import Existing Account”:



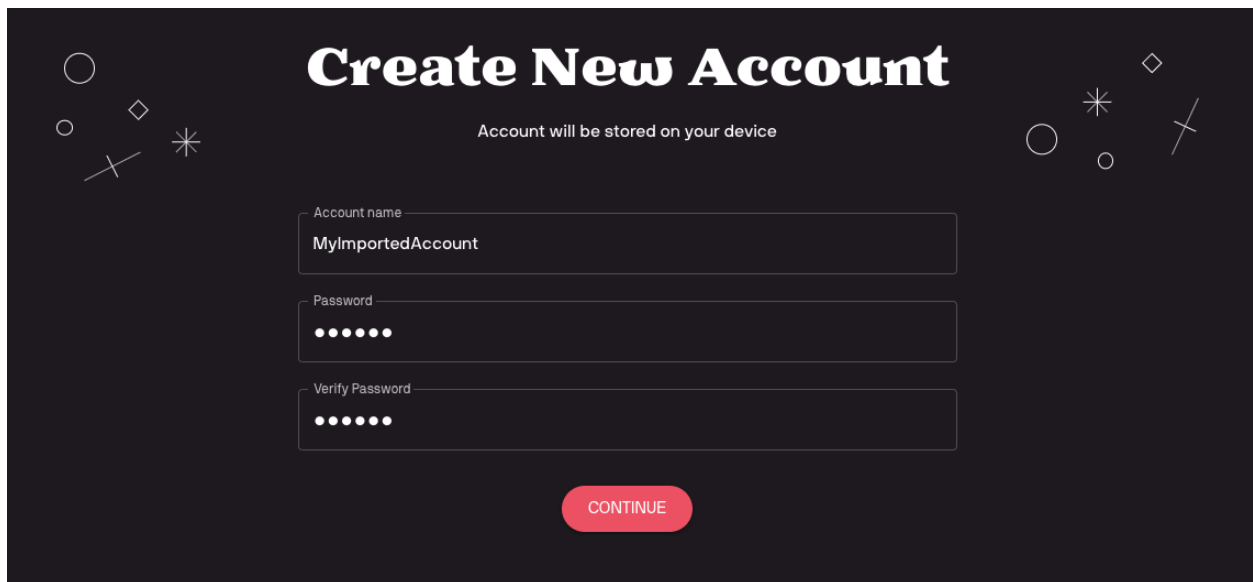
## Importing existing account

Import existing account feature is used when you are trying to access your account on other devices, or when you have forgotten your password so can't login normally.

Importing account is a 2-steps process.



1. First user will be asked to provide the 15 words mnemonic from when they created that account in correct order.



2. On the next screen user will be asked to provide a name for the account and choose the password for it, which will be used to access the account on that device from now on.

## Dashboard

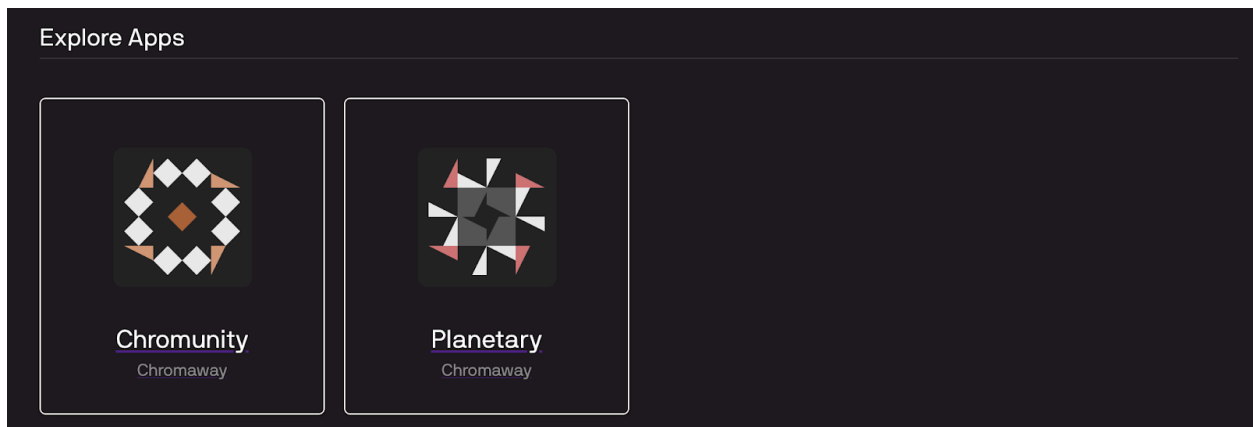
The Dashboard page is separated into 3 different sections: Chromia Accounts, Linked Apps and Explore Apps.

## Chromia accounts



Chromia Accounts are something that we usually call “main chain” accounts. There could be multiple Chromia Accounts within one Vault Account. Gaining access to a Vault Account will allow access to all Chromia Accounts beneath it.

## Explore Apps



Explore Apps section is basically app explorer (Google play / App store equivalent) where one can browse and explore all the apps built in Chromia ecosystem.

## Linked Apps



Linked Apps section contains all the apps that user has created an account for, which this Vault Account has control over.

Each dapp in Chromia ecosystem has its own blockchain. Every account (that little “Tile”) on the dashboard represents a combination of blockchain and specific account on that particular blockchain. These “Tile” are composed of 2 parts:

- Automatically generated squared image created from Blockchain ID, and
- Automatically generated robo-icon created from the Account ID.

That means if you have multiple accounts on the same Blockchain, they will have the same squared image, while robo-icon will be the same for the one accountId on different dapps.

Clicking on any of the accounts in the dashboard is taking to the “wallet” functionality of the Vault, used to send/receive assets from/to that specific account.

### Asset Transfer (Wallet features)

In order to access the wallet section, one needs to select the account from the dashboard first:



## Assets

Assets	
Name	Amount
CHROMA	227

Assets section is showing the list of all the available assets in that specific Account.

## Sending assets

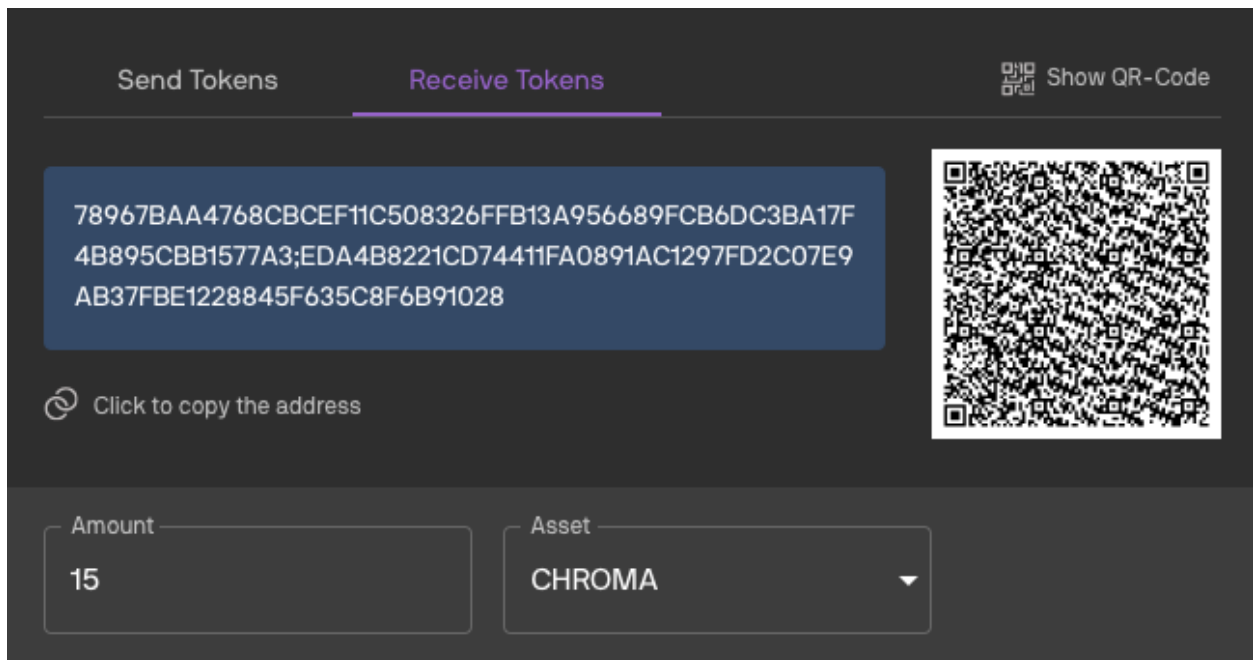
Assets are sent from the “Send Tokens” tab of the transfer section.

You can either enter the address manually (copy/paste) or use a QR scanner to scan the QR code containing recipient address info. The address is composed of 2 parts - blockchain id and account id, separated by the semicolon. So, address format is <blockchainId;accountId>.

Once address field is populated, “Application” and “Account” will be filled with appropriate hash icons automatically generated from the input address.

After address is populated, select an appropriate Asset and Amount to send, and click “Send”.

## Receiving assets



Send Tokens **Receive Tokens** Show QR-Code

78967BAA4768CBCEF11C508326FFB13A956689FCB6DC3BA17F  
4B895CBB1577A3;EDA4B8221CD74411FA0891AC1297FD2C07E9  
AB37FBE1228845F635C8F6B91028

Click to copy the address

Amount: 15 Asset: CHROMA

On the “Receive Tokens” tab of transfer section, the address for this specific account is shown.













There is also a QR code shown next to the address. Instead of providing the address itself, it’s possible to provide QR code which someone can scan and send assets.

Furthermore, besides holding just address info, QR code can also hold “Amount” and “Asset” information. Whenever “Amount” or “Asset” fields are changed, QR code is being updated.

When such a QR code (containing asset or amount info) is scanned from the “Send Tokens” tab, those fields will be auto-populated on the UI as well.

## Transaction history

Transaction History

Type	To/From	Asset	Amount	Timestamp	Copy Tx
 RECEIVED	 	CHROMA	22	18/11/2019, 09:00:25	
 RECEIVED	 	CHROMA	15	18/11/2019, 08:59:04	
 SENT	 	CHROMA	-10	18/11/2019, 08:58:47	

Rows per page: 5

1-3 of 3

|<

<

>

|>

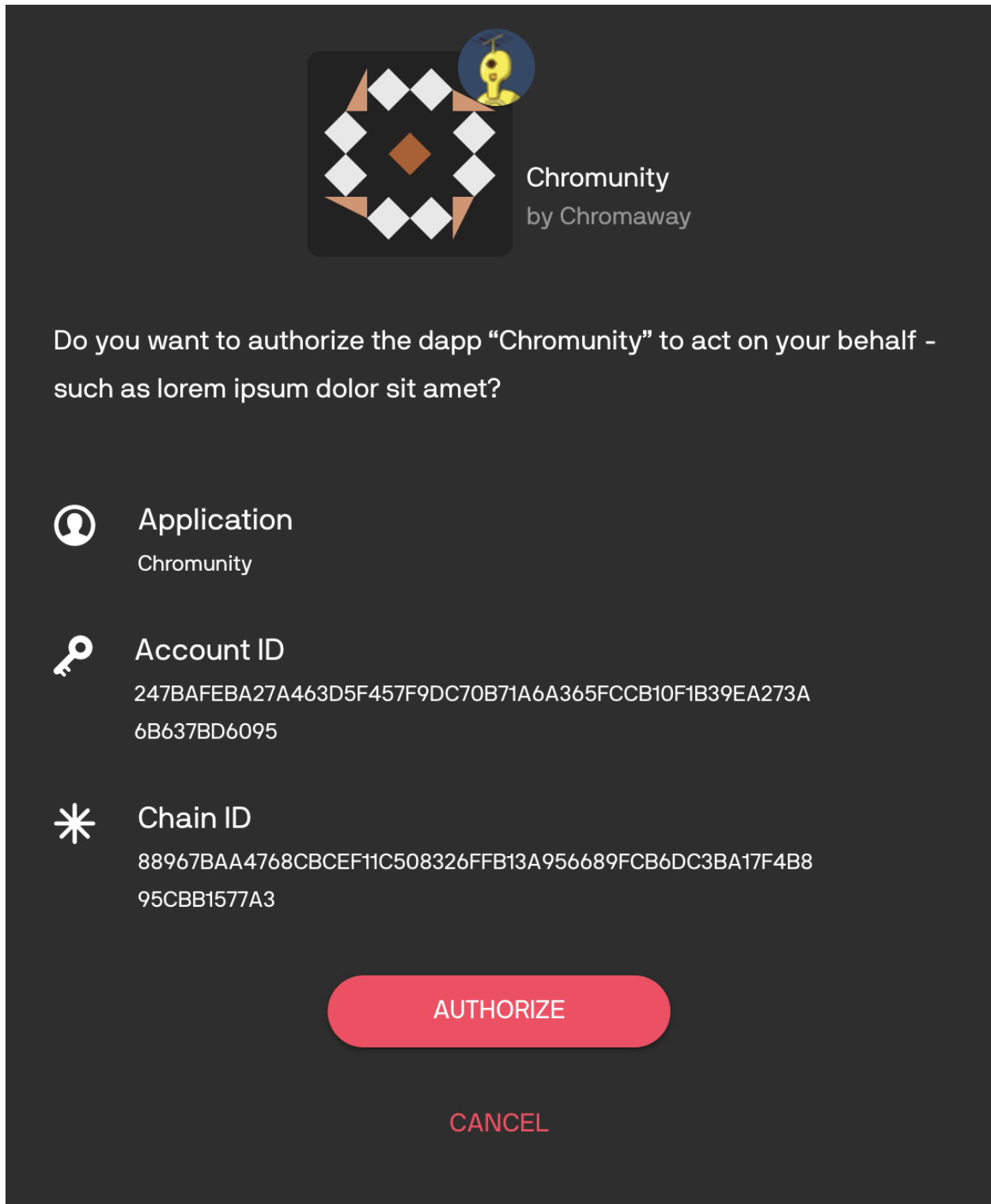
Transaction history is a table showing account’s transactional activities.

It contains info about transaction type (sending or receiving), the account to which we have sent or from which we have received the assets (sender/recipient), along with some other information like which Assets were transferred, the amount, timestamp, etc.

## **SSO and app linking**

Chromia Vault offers Single Sign-On (SSO) service for the dapps in Chromia ecosystem. This allows users to login to different systems (apps) using single account. In order to take advantage of it, the application needs to integrate with Chromia Vault SSO service. Similarly to “Login with Facebook” or “Login with Google” features, once Chromia Vault has been used for SSO, user will have to authorize the app in the system. That looks like on the image below:





### 2.6.3 FT3 Module

FT3 is the recommended standard to handle accounts and tokens in the Chromia ecosystem. It allows dapps to make full use of Chromia Vault, including Single Sign-On (SSO), asset transfers and visibility on the Vault’s dapp explorer.

FT3 consists of a Reli module that contains blockchain logic, and a client side library that provides a JS API for interaction with the module.

## Features

FT3 supports the following features:

- Account management
- Asset management and transaction history
- Single Sign-on (SSO) from the Vault

## Account Management

The central entity of FT3 module is `account`. An account is uniquely identified by an `id`:

```
entity account {  
  key id: byte_array;  
}
```

Account can be controlled by multiple users with different level of access rights. The authentication descriptors defines who can control an account and what he can do with it:

```
entity account_auth_descriptor {  
  descriptor_id: byte_array;  
  key account, descriptor_id;  
  index descriptor_id;  
  auth_type: text;  
  args: byte_array;  
}
```

At the moment, the module defines two types of authentication descriptors: **SingleSig** and **MultiSig** and two authorization types: **Authentication** ("A") and **Transfer** ("T"). The A flag specifies who can edit an account (and thus has all privileges to the account), and the T can only transfer assets.

Although there are only two predefined authorization flags, dapp developers are free to add more flag types to create a custom access control for his dapp.

**SingleSig** authentication descriptor is used to provide access to a single user. The descriptor accepts user's public key and authorization flags which specify what access rights the user has:

```
struct single_sig_args {  
  flags: set<text>;  
  pubkey;  
}
```

**MultiSig** authentication descriptor provides M of N control of an account. It accepts a list of N public keys, of which a minimum number M of signatures are required to authorize an operation and a set of authorization flags:

```
struct multi_sig_args {  
  flags: set<text>;  
  signatures_required: integer;  
  pubkeys: list<pubkey>;  
}
```

## Asset Management

FT3 provides support for multiple assets. The `asset` table contains list of registered assets:

```
entity asset {
  id: byte_array;
  key id;
  key name;
  issuing_chain_rid: byte_array;
}
```

**Note:** Although we can only transfer within the same chain for now, `issuing_chain_rid` is kept in preparation for coming release when FT3 support cross-chain asset transfer.

The `balance` table keeps track of an account's assets:

```
entity balance {
  key account, asset;
  mutable amount: integer = 0;
}
```

## Single Sign-on (SSO)

SSO allows a user to login to different applications with a single account (similar to how “Login with Google/Facebook” work).

The *Vault* is an SSO service of the Chromia ecosystem: any FT3 dapp can be configured to allow users to login using their Vault account.

## Project Setup

In this section, we explain how to setup a project to use FT3.

First let's clone FT3's bootstrap project repository:

```
git clone https://bitbucket.org/chromawallet/develop-chromia.git
```

Create a new directory for your project, and copy the `postchain` and `Reli` directories over to your project. The remaining `client` contains an example for Single Sign-On feature, so we will come back to it later.

## Blockchain side setup

### Config dapp description

1. Go to `postchain/config/nodes/`, you will find a `dev` directory. This is our postchain config directory. You can rename it to whichever name matching your convention (e.g. `prod` or `dapp_name`)
2. Inside `dev` directory, open the file `blochains/app/config.template.xml`, and change the settings for your chain:

```
<run wipe-db="true">
  <nodes>
    <config src="../../node-config.properties" add-signers="true" />
  </nodes>
  <chains>
    <chain name="YOUR_CHAIN_NAME" iid="0">
      <config height="0">
        <app module="">
          <args module="lib.ft3.core">
            <arg key="my_blockchain_name"><string>YOUR_DAPP_NAME</string></
↳arg>
            <arg key="my_blockchain_website"><string>YOUR_DAPP_WEBSITE</
↳string></arg>
            <arg key="my_blockchain_description"><string>YOUR_DAPP_
↳DESCRIPTION</string></arg>
            <arg key="rate_limit_active"><int>1</int></arg>
            <arg key="rate_limit_max_points"><int>10</int></arg>
            <arg key="rate_limit_recovery_time"><int>30000</int></arg>
            <arg key="rate_limit_points_at_account_creation"><int>1</int></
↳arg>
          </args>
        </app>
      </config>
    </chain>
  </chains>
</run>
```

**my\_blockchain\_name** Name of your chain.

**my\_blockchain\_website** “Main page” url of your dapp.

**my\_blockchain\_description** Description of your dapp.

The following arguments is settings for the rate limiter (spam prevention). The client will accumulate one “operation point” every `rate_limit_recovery_time` milliseconds, up to `rate_limit_max_points`. One point is spent for each operation.

**rate\_limit\_active** 0 for not active (no spam prevention) or 1 to activate the rate limit.

Note that even if rate limiter is not active, you must set some values in the following args.

**rate\_limit\_max\_points** Maximum amount of operation points that is possible to accumulate (and therefore the maximum number of transactions that can be made at once)

**rate\_limit\_recovery\_time** (In milliseconds) period of cool down before an account can receive one operation point.

**rate\_limit\_points\_at\_account\_creation** How many points an account have at the moment of creation (0 is min)

---

**Note:** Please note that if you use Single Sign-On, an account need to perform 1 operation immediately at the moment of creation to add disposable `auth_descriptor` (eg. SSO need `rate_limit_points_at_account_creation` at minimum of 1).

Refer to the *SSO Section* for more information.

---

## Database setup

We have provided a docker image of the database for ease of use:

```
cd postchain
docker image pull chromaway/postgres:2.4.0-beta
docker-compose up
```

**Note:** If you don't want to use Docker, or want to setup your own database, please follow the instructions in [Database Setup](#).

Update the `postchain/config/nodes/dev/node-config.properties` file to match your database settings.

## Running the chain

(From root directory) start your chain:

```
./postchain/bin/run-node.sh dev
```

Replace `dev` with the name of your postchain config directory if you have renamed it.

If everything is properly configured, you will soon see a success message printed to the console:

```
Postchain node launching is done
```

Above that line you will find the generated blockchain ID of the blockchain that looks like this:

```
INFO 2120-01-01 23:59:59.999 [main] BaseConfigurationDataStore - Creating initial_
↳ configuration for chain 1 with BC RID: _
↳ B61EFF348B43D7C93F67F6D2ABE17391D709A77F9A040D6309984665082DFE8A
```

Note down the blockchain ID, we will use it to connect to the chain.

**Important:** Postchain will generate a blockchain ID for dapp based on the dapp's codebase.

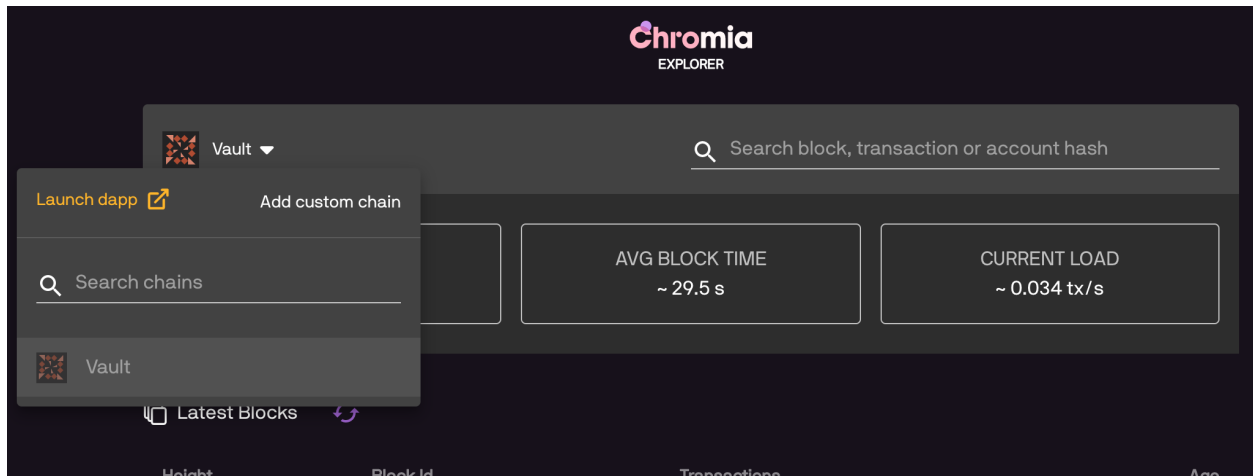
Whenever you change blockchain code of dapp, you will need to wipe database by adding the `-W` option, in order to get new blockchainID:

```
postchain/bin/run-node.sh dapp_name -W
```

If you missed the log in console, you can always check previous log in `logs/logfile.log` file.

## Verify the chain is working

Go to the [Chain Explorer](#). Click the dropdown next to "Vault", then choose "add custom chain":



In the following popup, enter your chain's information, using information you entered in `config.template.xml` and the chain BRID:

Add chain

Chain Name

My dapp

Host

http://localhost

Port

7743

DApp website

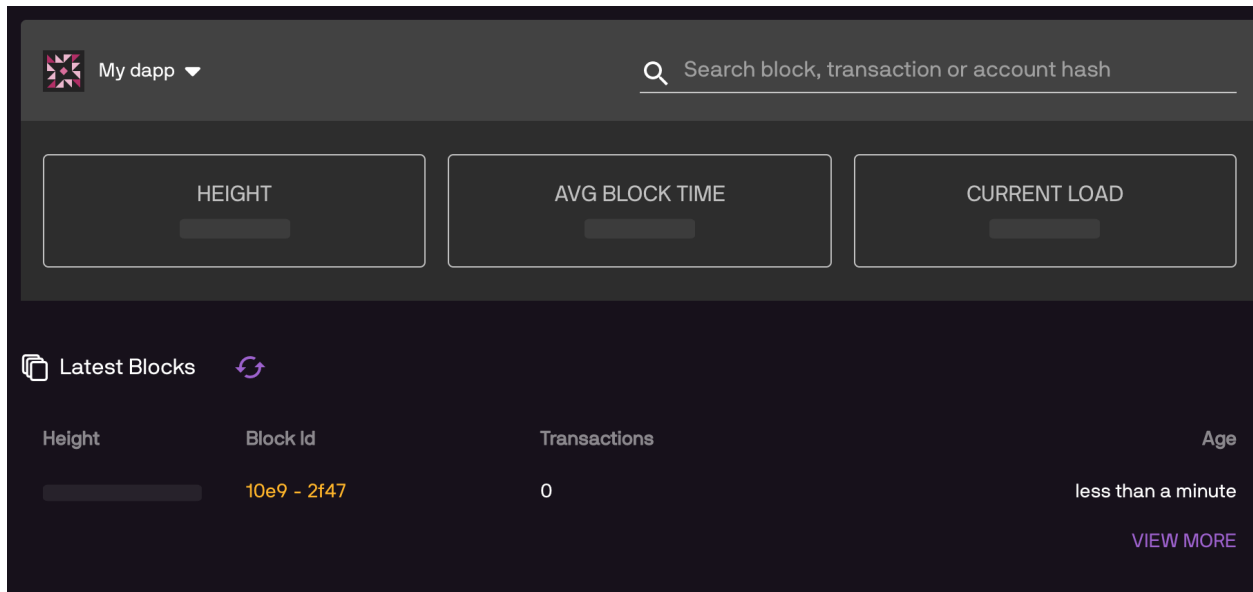
http://localhost:3000

Chain ID (RID)

2C6AC2C607912185AD9775F9BFD8809F61FA235D0EB89910EAFE25465

ADD CHAIN

If you see the chain's information displayed, then your chain is working properly:



If chain explorer can't connect to your chain, it indicates something is wrong with your settings from previous steps. Verify that the host and ports are correct (7743 is the default port from `node-config.properties`), and also your blockchainID.

With that the blockchain side is ready, we can go on to the client side.

## Client side setup

The `client` directory you use in the bootstrap project is an example client, which will work with our current chain. In this section, we will discuss how to create our own client that connects to the chain.

Create a `client` directory for your project, run `npm init` (or bootstrap a project using a generator, e.g. `create-react-app`).

Add dependencies to the nodejs project:

```
npm i --save ft3-lib
npm i --save postchain-client
```

Add other libraries to your liking.

## Set config variables

Choose your own method to set these important config variables:

```
export const blockchainRID = "<YOUR CHAIN BRID>";
export const blockchainUrl = "http://localhost:7743/"; // This is default value in
node-config.properties file
export const vaultUrl = "https://dev.vault.chromia-development.com"; // Vault's url
for SSO
```

That concluded the project setup process. In next section, we will continue working with the client library and discuss the features of `ft3-lib` npm package.

## Javascript library

In this section, we explain how to use the client side library (ft3-lib node package).

### Initialize Blockchain object

The first thing that has to be done before a blockchain can be accessed is to initialize the Blockchain object used to interact with the blockchain:

```
// /client/index.js
import { Postchain } from 'ft3-lib';
import { blockchainRID, blockchainUrl } from './configs/constants'; // these configs
    ↳ are set in previous section

const chainId = Buffer.from(blockchainRID, 'hex');
const blockchain = await new Postchain(blockchainUrl).blockchain(chainId);
```

Details of the initialized chain can be accessed from info property which has name, website and description properties:

```
console.log(`----- Blockchain Info -----`);
console.log(`name      : ${blockchain.info.name}      `);
console.log(`website    : ${blockchain.info.website}    `);
console.log(`description : ${blockchain.info.description} `);
```

These fields have the values which we previously set in the config.template.xml file.

---

## The User class

The User class represents a logged in user, and is used to keep the user's key pair and authentication descriptor.

Any method that require transaction signing will need an object of this class.

```
import { SingleSignatureAuthDescriptor, User, FlagsType } from 'ft3-lib';

...

const authDescriptor = new SingleSignatureAuthDescriptor(
    keyPair.pubKey,
    [FlagsType.Account, FlagsType.Transfer]
);
const user = new User(keyPair, authDescriptor);
```

Many functions provided by Blockchain class require User object, for example:

```
const authDescriptor = ...;
const user = ....;

// gets all accounts where this authDescriptor has control over
const accounts = await blockchain.getAccountsByAuthDescriptorId(
    authDescriptor.id,
    user
);
```



In most of the cases the same User instance is used throughout an app (the “current user”). In order to avoid passing both Blockchain and User objects around in an app, the *BlockchainSession* class is introduced.

It has many of the same functions as Blockchain class, but with a difference that functions provided by the BlockchainSession don’t require User parameter:

```
const authDescriptor = ...;
const user = ....;

const session = blockchain.newSession(user);
const accounts = await session.getAccountsByAuthDescriptorId(authDescriptor.id);
```

## AuthDescriptor Rules

Both AuthDescriptor constructors accept an optional 3rd parameter of type Rules, which define constraint for the descriptor’s “valid period”.

Supported constrains are:

**Rules.operationCount** Number of operations this authDescriptor can perform:

```
import { SingleSignatureAuthDescriptor, FlagsType, Rules } from 'ft3-lib';

const authDescriptor = new SingleSignatureAuthDescriptor(
  keyPair.pubKey,
  [FlagsType.Account, FlagsType.Transfer],
  Rules.operationCount.lessOrEqual(2) // This authDescriptor is only valid for 2_
  ↳operations
);
```

**Rules.blockTime** Time period during which the authDescriptor has effect:

```
import { SingleSignatureAuthDescriptor, FlagsType, Rules } from 'ft3-lib';

const authDescriptor = new SingleSignatureAuthDescriptor(
  keyPair.pubKey,
  [FlagsType.Account, FlagsType.Transfer],
  Rules.blockTime.greaterThan(Date.now() + 12 * 60 * 60 * 1000) // This_
  ↳authDescriptor will start working 12 hours from now
);
```

**Rules.blockHeight** Block heigh limitation of the authDescriptor:

```
import { SingleSignatureAuthDescriptor, FlagsType, Rules } from 'ft3-lib';

const authDescriptor = new SingleSignatureAuthDescriptor(
  keyPair.pubKey,
  [FlagsType.Account, FlagsType.Transfer],
  Rules.blockHeight.equal(0) // This authDescriptor is only valid when the chain was_
  ↳just created (0 block is in the chain)
);
```

Supported operators are:

- lessThan
- lessThanOrEqual
- equal

- greaterThan
- greaterOrEqual

Is it also possible to build composite rules:

```
import { SingleSignatureAuthDescriptor, FlagsType, Rules } from 'ft3-lib';

// This authDescriptor will start working 12 hours from now and is only valid for 24
↪hours
const startDate = Date.now() + 12 * 60 * 60 * 1000;
const endDate = Date.now() + 36 * 60 * 60 * 1000;
const authDescriptor = new SingleSignatureAuthDescriptor(
  keyPair.pubKey,
  [FlagsType.Account, FlagsType.Transfer],
  Rules.blockTime.greaterThan(startDate).and.blockTime.lessThanOrEqual(endDate)
);
```

### The Account class

An Account object contains:

- assets: an array of AssetBalance instances.
- authDescriptor: an array of AuthDescriptor instances.
- session: the BlockchainSession that returned it.

### Account registration

```
const ownerKeyPair = ...;
const authDescriptor = new SingleSignatureAuthDescriptor(
  ownerKeyPair.pubKey,
  [FlagsType.Account, FlagsType.Transfer]
);

const account = await blockchain.registerAccount(authDescriptor, user);
```

More commonly the current user will be creating an account for themselves. In those cases we can simply pass `user.authDescriptor` into the operation:

```
const account = await blockchain.registerAccount(user.authDescriptor, user);
```

### Searching accounts

Accounts can be searched by account ID:

```
const account = await session.getAccountById(accountId);
```

by authentication descriptor ID:

```
const accounts = await session.getAccountsByAuthDescriptorId(authDescriptorId);
```

or by participant ID:

```
const accounts = await session.getAccountsByParticipantId(user.keyPair.pubKey);
```

For SingleSig and MultiSig account descriptors, participant ID is pubKey. Therefore this function allows to search for accounts by pubKey.

**Important:** The difference between `getAccountsByParticipantId` and `getAccountsByAuthDescriptorId` is:

- `getAccountsByParticipantId` returns all accounts where user is participant, no matter which access rights user has or which type of authentication is used to control the accounts
- while `getAccountsByAuthDescriptorId` returns only accounts where user has access with specific type of authentication and authorization.

## Adding authentication descriptor

```
const newAuthDescriptor = new SingleSignatureAuthDescriptor(
  pubKey,
  [FlagsType.Account, FlagsType.Transfer]
);
const account = await session.getAccountById(accountId);
await account.addAuthDescriptor(newAuthDescriptor);
```

## Assets Management

### AssetBalance

Each account when queried comes with an `account.assets` array of `AssetBalance`.

An `AssetBalance` contain information about an Asset the account own (`assetBalance.asset`), and the amount owned (`assetBalance.amount`).

You can also get asset balance of an account by calling `AssetBalance.getByAccountId`:

```
const balances = await AssetBalance.getByAccountId(accountId, blockchain);
```

or if you are interested in only one specific asset:

```
const balance = await AssetBalance.getByAccountAndAssetId(accountId, assetId, ↵
↵blockchain);
```

### Asset

Each Asset has a name and a `chainId`, which is the id of the chain where the asset come from.

The unique identifier of asset (`asset.id`) is generated from the hash of name and `chainId`, so asset name is unique within a chain but different chains can have asset with the same name.

An Asset must be registered on a chain to be recognized:

```
await Asset.register(assetName, blockchainId, blockchain);
```

Registered assets can be queried by name:

```
const assets = await blockchain.getAssetsByName(assetName);
```

or by id:

```
const asset = await blockchain.getAssetById(assetId);
```

You can also get all assets of a chain by calling `getAllAssets`:

```
const assets = await blockchain.getAllAssets();
```

## Transferring assets

```
const account = await session.getAccountById(accountId);  
await account.transfer(recipientId, assetId, amount);
```

---

**Note:** Here we see that the `Account` class retains the same characteristic as `BlockchainSession`: we don't need to provide an `User` object to sign the transaction.

---

## Transfer History

```
const history = await account.getPaymentHistory();
```

`getPaymentHistory` return an array of `PaymentHistoryEntryShort`:

```
class PaymentHistoryEntryShort {  
  readonly isInput: boolean; // true if account is the sender, false in case of_  
  ↪ receiver  
  readonly delta: number; // amount transferred: negative for sender (e.g. -10),_  
  ↪ positive for receiver (e.g. 12)  
  readonly asset: string;  
  readonly assetId: string;  
  readonly entryIndex: number;  
  readonly timestamp: Date;  
  readonly transactionId: string;  
  readonly transactionData: Buffer;  
  readonly blockHeight: number;  
}
```

## Calling operations

### Single operation

FT3 operations and other blockchain operations can also be directly called using the `Blockchain` and `BlockchainSession` classes.

For instance, the same “adding auth descriptor” operation above can be done using:

```
import { op } from 'ft3-lib';

const account = ...
const user = ...
const newAuthDescriptor = ...

await blockchain.call(
  op(
    'ft3.add_auth_descriptor',
    accountId,
    user.authDescriptor.id,
    newAuthDescriptor
  ),
  user
)
```

## Multiple operations

The transaction builder can be used if multiple operations have to be called in a single transaction:

```
await blockchain.transactionBuilder()
  .add(op('foo', param1, param2))
  .add(op('bar', param))
  .buildAndSign(user)
  .post();
```

Previous statement creates a single transaction with both `foo` and `bar` operations, adds signers from user’s auth descriptor and signs it with user’s private key.

If more control is needed over signers and signing then `build` and `sign` functions could be used instead:

```
await blockchain.transactionBuilder()
  .add(op('foo', param1, param2))
  .add(op('bar', param))
  .build(signersPublicKeys)
  .sign(keyPair1)
  .sign(keyPair2)
  .post();
```

Instead of immediately sending a transaction after building it, it is also possible to get a raw transaction:

```
const rawTransaction = blockchain.transactionBuilder()
  .add(op('foo', param1, param2))
  .buildAndSign(user)
  .raw();
```

which can be sent to a blockchain node later:

```
await blockchain.postRaw(rawTransaction);
```

## The nop operation

To prevent replay attack postchain rejects a transaction if it has the same content as one of the transactions already stored on the blockchain. For example if we directly call `ft3.transfer` operation two times, the second call will fail.

```
const inputs = ...
const outputs = ...
const user = ...

// first will succeed
await blockchain.call(op('ft3.transfer', inputs, outputs), user);

// second will fail
await blockchain.call(op('ft3.transfer', inputs, outputs), user);
```

To avoid transaction failing, nop operation can be added to a second transaction in order to make it differ from the first transaction.

```
import { op, nop } from 'ft3-lib';

await blockchain.transactionBuilder()
  .add(op('ft3.transfer', inputs, outputs))
  .add(nop())
  .buildAndSign(user)
  .post();
```

nop() function returns nop operation with a random number as argument.

## GtvSerializable interface

In typescript, op function is defined as:

```
function op(name: string, ...args: GtvSerializable[]): Operation {
  return new Operation(name, ...args);
}
```

It expects arguments to implement GtvSerializable interface, i.e. to have implemented toGTV() function.

Array, Buffer, String and Number are already extended with toGTV function.

If user defined object wants to be passed to an operation, it has to implement GtvSerializable interface, e.g.

```
class Player {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  toGTV() {
    return [this.firstName, this.lastName],
  }
}

await blockchain.call(
  op('some_op', new Player('John', 'Doe')),
  user
)
```

To be able to handle the Player object, on blockchain side, some\_op would have to be defined as either:

```
operation some_op(player: list<gtv>) {
    ...
}
```

or

```
struct player {
    first_name: text;
    last_name: text;
}

operation some_op(player) {
    ...
}
```

## Rell Integration

In the bootstrap project, `rell/src` directory is configured to be the Rell project's root module. `rell/src/lib/ft3` directory contains the FT3 module.

We also provided a `rell/src/module.rell` file with some template code which could serve as an entry point for your dapp, although you are free to choose any module structure to your liking.

## Defining dapp\_account entity

The `account` entity provided by `.lib.ft3.account` module can be used for account management. But your dapp might need to record more information about the user, like username, email address, etc. We can add an entity to record those information.

An example dapp's account could be defined as:

```
import acc: .lib.ft3.account;

entity dapp_account {
    key account: acc.account;
    key username: text;
}
```

In this user model, there is no public key or user ID. Those details are provided by `acc.account`.

`acc.account` has an `id` property which uniquely identifies an account, and access is controlled by `acc.account_auth_descriptor` which include user's public key.

The underlying structure of `acc.account` and `acc.account_auth_descriptor` is explained in [Account Management](#).

After `dapp_account` entity is defined, the next step is to define operation used to create an instance of `dapp_account`:

```
operation create_dapp_account (
    username: text,
    user_auth: acc.auth_descriptor
) {
    val account_id = acc.create_account_with_auth(user_auth);
    create dapp_account (
```

(continues on next page)

(continued from previous page)

```

    username,
    acc.account @ { account_id }
  );
}

query get_dapp_accounts() {
  return dapp_account @? {} ( id = dapp_account.account.id, first_name = .first_name,
  ↪last_name = .last_name, age = .age);
}

```

Restart the node for changes to take effect. Because we have changed database structure, we need to add `-W` option to delete the database and add new `dapp_account` table:

```
postchain/bin/run-node.sh <dapp_name> -W
```

**Important:** Make sure to update your client's blockchainRID config with the newly generated blockchainRID

On the client side, operation can be called using `ft3-lib` (or `postchain-client`):

```

import DirectoryService from './lib/directory-service';
import { util } from 'postchain-client';
import { blockchainRID } from '../configs/constants';

import {
  op,
  Blockchain,
  SingleSignatureAuthDescriptor,
  FlagsType,
  User
} from 'ft3-lib';

const keyPair = util.makeKeyPair();
const user = new User(
  keyPair,
  new SingleSignatureAuthDescriptor(
    keyPair.pubKey,
    [FlagsType.Account, FlagsType.Transfer]
  )
);

const blockchain = await Blockchain.initialize(
  blockchainRID,
  new DirectoryService()
);
const session = blockchain.newSession(user);

await session.call(op(
  'create_dapp_account',
  'John',
  'Doe',
  30,
  user.authDescriptor
));

```

We can check if `create_dapp_account` operation is executed successfully using `get_dapp_accounts` query



we created:

```
const rest = pcl.restClient.createRestClient(nodeApiUrl, blockchainRID, 5)

const gtx = pcl.gtxClient.createClient(
  rest,
  Buffer.from(
    blockchainRID,
    'hex'
  ),
  [],
);

const allDappAccoounts = await gtx.query('get_dapp_accounts');
```

## Modules

In this section, we will go through some of the built-in utilities that FT3 modules provide.

It should be noted that the built-in operations and queries all have matching interface in `ft3-lib` *javascript library*.

### Account Module

```
import acc: .lib.ft3.account;
```

#### Functions:

```
function create_account_with_auth (auth_descriptor): bytearray
```

Create a new FT3 account using the provided `ft3.account.auth_descriptor`

- `auth_descriptor`: The `auth_descriptor` used to create this account.
- `return`: `account.id` (equal to `auth_descriptor.hash()`)

```
function auth_and_log(account_id: bytearray, auth_descriptor_id: bytearray,
↳required_flags: list<text>): account
```

Authorize given `auth_descriptor` for required authorization flags, and apply the rate limiter constrains configured in `config.template.xml`. This is meant to be the default authorization mechanism for operations.

- `account_id`: id of the account
- `auth_descriptor_id`: is equal to `auth_descriptor.hash()`
- `required_flags`: list of required authorization flags (see *Account Management*)
- `return`: the account instance

```
function require_auth (account, descriptor_id: bytearray, required_flags: list<text>)
```

Authorize given `auth_descriptor`, but does not apply rate limiter's constrains.

```
function _add_auth_descriptor (account, auth_descriptor)
function _delete_auth_descriptor(auth_descriptor: account_auth_descriptor)
function _delete_all_auth_descriptors_exclude(account, auth_descriptor_id: bytearray)
```

Utilities for managing auth\_descriptors.

**Operations:**

```
operation delete_auth_descriptor (account_id: bytearray, auth_descriptor_id: bytearray
↳array, delete_descriptor_id: bytearray)

operation delete_all_auth_descriptors_exclude(account_id: bytearray, auth_descriptor_
↳id: bytearray)

operation add_auth_descriptor (account_id: bytearray, auth_id: bytearray, new_desc:
↳acc.auth_descriptor)
```

**Queries:**

```
query get_account_auth_descriptors(id: bytearray)

query get_account_by_id(id: bytearray)

query get_account_by_auth_descriptor(auth_descriptor)

query get_accounts_by_participant_id(id: bytearray)

query get_accounts_by_auth_descriptor_id(descriptor_id: bytearray)
```

**Core Module**

```
import core: .lib.ft3.core;
```

**Functions:**

```
function register_asset (name, issuing_chain_id: bytearray): asset
```

Register a new asset on the chain.

```
function _get_asset_balances(account_id: bytearray): list<(id:bytearray,name:text,
↳amount:integer,chain_id:bytearray)>
```

Get asset balance of an account.

```
function ensure_balance(acc.account, asset): balance
```

Get account's balance of an asset, or create one if it doesn't exist.

```
struct xfer_input {
  account_id: bytearray;
  asset_id: bytearray;
  auth_descriptor_id: bytearray;
  amount: integer;
  extra: map<text, gtv>;
}

struct xfer_output {
  account_id: bytearray;
  asset_id: bytearray;
  amount: integer;
  extra: map<text, gtv>;
}
```

(continues on next page)

(continued from previous page)

```

}

function _transfer (inputs: list<xfer_input>, outputs: list<xfer_output>)

```

Perform an asset transfer from accounts described in `xfer_input` to accounts in `xfer_output`.

If `xfer_output.extra` map contains a `reg_auth_desc` key, then the value will be used as `auth_descriptor` to create a new account (meaning you can create a new account then transfer asset to it immediately in one transaction).

#### Operations:

```

operation transfer (inputs: list<ft3.xfer_input>, outputs: list<ft3.xfer_output>)

```

#### Queries:

```

query get_asset_balances(account_id: byte_array)

query get_asset_balance(account_id: byte_array, asset_id: byte_array)

query get_asset_by_name(name)

query get_asset_by_id(asset_id: byte_array)

query get_all_assets()

query get_payment_history(account_id: byte_array, after_block: integer)

```

## Single Sign-on (SSO)

SSO allows user to login to different dapps with a single account. It is similar to how you can click a “Login with Facebook/Google” buttons to login to different services using a single Facebook/Google account.

In order for SSO to work, dapp must be integrated with a SSO service. *Chromia Vault* is the main SSO service in Chromia ecosystem, though a custom SSO service can be easily implemented thanks to FT3 module’s flexible authentication model.

As discussed in previous sections, access to a FT3 account is controlled with authentication descriptors (See *Account Management*). So if we use a Vault account’s public key to create an `authDescriptor` with 'A' flag, and add it to a dapp account, Vault will have control over the dapp account.

The actual process is done in 3 steps:

- User login to their Vault account, and approve of the SSO request.
- Vault’s `keyPair` is used to create a new account for dapp (so dapp account and Vault account will have the same `accountId`). A second `authDescriptor` with only 'T' flag is created on-the-fly and added to this new dapp account.
- The user can now use the second `keyPair` to perform transactions for the account.

This `keyPair` is meant to be disposable and can safely be discarded or replaced at user’s discretion.

This might sound rather complicated to implement, but fortunately `ft3-lib` library already handle all of the heavy works, and dapps only have to config the SSO Class a bit for it to work.

The bootstrap project already contain a `client` directory that implement the SSO flow. Let’s go through how to set up the flow using bootstrap client, then we will discuss how to config SSO flow for your own client.

## SSO flow with bootstrap client

Backup your `client` directory somewhere safe, and replace it with the `client` directory from bootstrap project.

## Set config variables

Bootstrap client use `dotenv` npm package to set config variables. In `client` directory you can see a `.env.sample` file. Duplicate the file and rename it to `.env`. Update the file with your chain's settings, be sure to uncomment those lines (remove leading `#`):

```
REACT_APP_VAULT_URL=https://vault-testnet.chromia.com
REACT_APP_BLOCKCHAIN_RID=DAPP_BLOCKCHAIN_RID
REACT_APP_NODE_ADDRESS=http://localhost:7743
```

## Setup the dapp on Vault

Because our dapp is not public on Chain Explorer yet, we need to config it for testing as a Custom DApp.

Go to the [Vault page](#) and login to your account (Follow the instruction at [Chromia Vault](#) section if you are unsure).

Scroll down to the “All DApps” section, and click “Add Custom DApp”. Fill in information of your chain (similar to how you did with Chain Explorer during project setup).

Save changes

×

DApp Name

Some Name

Host

http://localhost

Port

7743

Website

http://localhost:3000

Chain ID (RID)

1D0212D01C93F69DB465F37703A5A96C7C806EE865588A60A566BDF3

REMOVE DAPP

SAVE CHANGES

You will see your dapp tile added to the Vault's dapp list.

Now the SSO flow is ready to use. Go back to your dapp (click the tile in Vault's dapp list), and you will see a login screen. Clicking the login button will redirect you to Vault to login to your Vault account.

If everything was setup correctly, Vault will ask you to authorize dapp. Clicking "Authorize" button will redirect you back to your client, with the newly created account's information displayed.

If the Authorize page is not displayed, it indicates a problem with your configs. Verify that the BRID and host is correct and try again.

We got the bootstrap client to work with SSO. You can use the bootstrap client as a base to build your own client. But if you want to implement SSO in your own client, the next part discuss how to do exactly that.

## SSO Class

FT3 provides SSO functionality through SSO class. The first step in integrating SSO into a dapp is to initialize SSO class on the app launch:

```
import { Blockchain, SSO } from 'ft3-lib';
import { blockchainRID, blockchainUrl, vaultUrl } from './configs/constants'; //
↳ these configs are set in project-setup section

const chainId = Buffer.from(blockchainRID, 'hex');
const blockchain = await new Postchain(blockchainUrl).blockchain(chainId);

SSO.vaultUrl = vaultUrl;
const sso = new SSO(blockchain);
```

## Initiate login

When user click the “Login with Vault” button, dapp should call `initiateLogin`:

```
const { SSO } from 'ft3-lib';

const successUrl = `${window.location.origin}/success`;
const cancelUrl = `${window.location.origin}/cancel`;

sso.initiateLogin(successUrl, cancelUrl);
```

`initiateLogin` navigates a user to the Vault, where they have to select one of their accounts in order to create a new dapp account or login to existing dapp account. After the user logs in to their vault account and authorizes login or account creation, they will be redirected back to the dapp to finalize login.

- `successUrl`: where to redirect to after a successful login
- `cancelUrl`: where to redirect to when user cancels the login

## Finalize login

If there were no errors, Vault will redirect the user to the location of `successUrl`.

Vault will add query parameters containing raw transaction - which needs to be signed by the dapp and posted to the blockchain.

Singing and posting is handled by the `finalizeLogin` method:

```
import { parse } from 'query-string';

const { rawTx } = parse(search); // extract rawTx query parameter

try {
  const [account, user] = await sso.finalizeLogin(rawTx);
} catch (error) {
  // handle error
}
```

`finalizeLogin` returns a tuple which contains ft3 account (Account instance) and ft3 user (User instance).

If an account doesn't already exist, call to `finalizeLogin` will create it, and if it was already created, only auth descriptor would be added to the account. Returned user object contains a key pair used to sign the transactions and an auth descriptor to authorize operations.

This same flow is used both for registering new dapp account and login to existing account.

### Auto-login (Remember Me)

By default, SSO uses an instance of `SSOStoreDefault` to keep track of account (account id) and user (key pair) details in memory. As soon as SSO instance is destroyed (e.g. page refresh), account and user details are lost.

But it is also possible to persist them to local storage. Details will be stored to local storage if SSO is initialized with `SSOStoreLocalStorage`:

```
import { SSO, SSOStoreLocalStorage } from 'ft3-lib';

const sso = new SSO(blockchain, new SSOStoreLocalStorage());
```

On app launch, the auto-login feature can be used to auto login user if they have already logged in previously:

```
const [account, user] = await sso.autoLogin();

if (account === null) {
  // redirect to login page
}
```

Like `finalizeLogin` method, `autoLogin` returns account and user objects.

Auto login will only work if SSO can find account id and keypair in `LocalStorage`, and if the `authDescriptor` which corresponds to stored key pair didn't expire.

If `autoLogin` returns null for account and user, then user should be redirected to a normal login page, where `initiateLogin` should be called to login using the Vault.

**Warning:** Storing the key pair inside `LocalStorage` is potentially a security risk, therefore `SSOStoreLocalStorage` should only be used if dapp doesn't require a high level of security.

In high security cases, dapp should implement its own method of preserving `user.keyPair` for future use.

### Logout

Call `logout` method to delete local cache and auth descriptor:

```
await sso.logout();
```

**Note:** Make sure you are using the same SSO instance which was used when calling `finalizeLogin`, because account and user details are stored in that object instance.

### Low level details

As briefed in the beginning, SSO flow creates an ft3 account on a dapp's chain which has the same `accountId` as the user's Vault account.

In the flow, Vault is responsible for building a transaction which creates the account and adds an `authDescriptor` to it. Once Vault passes the transaction to the dapp (via request parameters), dapp signs it and posts it to the blockchain by calling `finalizeLogin`.

This transaction will perform 2 operations, which add two auth descriptors to the account. One is added with a call to `register_account` operation and second is added with `add_auth_descriptor` operation.

- First `authDescriptor` is created using Vault public key and has 'A' and 'T' flags
- The second is created using a disposable public key (generated by `initializeLogin` method) with only “T” flag.

If an account with the same `accountId` as Vault’s account already exists on the blockchain, SSO will only add the second `authDescriptor` with disposable public key.

---

**Important:** Because the new account need to add the disposable auth descriptor immediately at creation time, it is required that you set a value equal or greater than 1 for `rate_limit_points_at_account_creation` in `config.template.xml` as we have noted during [Project Setup](#).

If you also need to record additional user information (as discussed below), the minimum value will need to be increased further.

---

## Dapp Account when using SSO

SSO can only create a ft3 account, but in most cases that is not enough to store dapp account details.

Using the same example `dapp_account` entity in [Reli Integration](#):

```
entity dapp_account {  
  key account: ft3.account;  
  key username: text;  
}
```

We can make some changes to support the SSO flow:

```
operation create_dapp_account (  
  username: text,  
  account_id: byte_array,  
  auth_descriptor_id: byte_array  
) {  
  val account = auth_and_log(  
    account_id,  
    auth_descriptor_id,  
    list<text>()  
  );  
  
  create dapp_account (account, username);  
}  
  
query get_dapp_account(account_id: byte_array) {  
  return dapp_account @? { .account.id == account_id } (  
    account_id = .account.id,  
    username = .username  
  );  
}
```

And then on the client side the login flow would look like this:



```
const [account, user] = await blockchain.finalizeLogin(tx);

const dapp_account = await blockchain.query(
  'get_dapp_account',
  { account_id: account.id }
);

const username = 'john_doe'; // Fake username

if (!dapp_account) {
  await blockchain.call(
    op('create_dapp_account', username, account.id, user.authDescriptor.id),
    user
  );
}
```

On the login success page, we check if the dapp account exists by calling `get_user` query. If `null` is returned, that means the account doesn't exist yet, so we have to create it.

In this oversimplified example, it is done by calling `create_user` operation with hard-coded info. In a real dapp, we would need a form where user can enter those account details.

Next time the user logs in, `get_user` query will return the dapp account, so the dapp account creations step will be skipped.

## How to update a dapp chain

Dapps evolve over time. And in this section, we discuss what to do in order to seamlessly update a dapp's chain to new version.

There are 2 approaches to update a chain:

- Specify different config entry points for old and new versions, or
- Reference compiled configuration of old version

## Project structure before update

For both approaches, we will use an example project with a basic ft3 structure.

We have all of our chain's code within `rell/src` folder:

```
rell/src
├── lib
│   └── ft3
│       ... (other optional libraries)
└── module.rell
```

And our `config.template.xml` should look like the following:

```
<run wipe-db="true">
  <nodes>
    <config src="../../node-config.properties" add-signers="true" />
  </nodes>
  <chains>
    <chain name="YOUR_CHAIN_NAME" iid="0">
```

(continues on next page)

(continued from previous page)

```
<config height="0">
  <app module="">
    <args module="lib.ft3.core">
      <!-- ... (your chain configs) -->
    </args>
  </app>
</config>
</chain>
</chains>
</run>
```

## Contents

- *How to update a dapp chain*
  - *Project structure before update*
  - *Update by specifying different entry points*
    - \* *Restructure the code*
    - \* *Adding new codes*
    - \* *Set the migration to new module*
    - \* *Test the chain*
  - *Update by referencing old configuration*
    - \* *What does run-node.sh do?*
    - \* *Backup old configuration*
    - \* *Adding new codes*
    - \* *Set the migration to new module*

## Update by specifying different entry points

Use this approach when you want to have a clear overview of old version's code.

**Note:** An example project to demonstrate this approach is available at branch `example/update-chain-keep-old-code` of `develop-chromia`:

```
git clone https://bitbucket.org/chromawallet/develop-chromia.git
git checkout example/update-chain-keep-old-code
```

## Restructure the code

As mentioned earlier, the target of this approach is to have clear overview of old code after several updates. So it might be considered good procedure to create different entry module for each chain version.

We will name each module with the current version tag, so that it will be clear what has been run in the past. After that, we will update the `config.template.xml` file by adding each version configuration and the height at which the changes will become effective.

The rell structure will then look like:

```
rell/src
├── v0_0_1
│   ├── lib
│   │   ├── ft3
│   │   └── ... (other optional libraries)
│   └── module.rell
```

So essentially we just created a folder `v0_0_1` inside `src` and moved the all the previous files there.

We will need to update the `config.template.xml` file to reflect this change. Note the change at `<app>` and `<arg>` tags:

```
<run wipe-db="true">
  <nodes>
    <config src="../../node-config.properties" add-signers="true" />
  </nodes>
  <chains>
    <chain name="YOUR_CHAIN_NAME" iid="0">
      <config height="0">
        <app module="v0_0_1">
          <args module="v0_0_1.lib.ft3.core">
            <!-- ... (your chain configs) -->
          </args>
        </app>
      </config>
    </chain>
  </chains>
</run>
```

**Note:** As a matter of information, the project's entry path is specified under `postchain/config/nodes/dev/blockchains/app/entry-file.txt`.

Confirm the new configuration is working properly by starting the chain:

```
./postchain/bin/run-node.sh dev
```

## Adding new codes

Now that we have restructured the code, we can upgrade the code. To do that we copy `v0_0_1` folder and rename it `v0_0_2`.

```
rell/src
├── v0_0_1
│   ├── lib
│   │   ├── ft3
│   │   └── ... (other optional libraries)
│   └── module.rell
└── v0_0_2
```

(continues on next page)

(continued from previous page)

```

├── lib
│   ├── ft3
│   └── ... (other optional libraries)
└── module.rell

```

For easy testing, we can add one new query in the `module.rell` file of `v0_0_2`:

```

module;

import lf: .lib.ft3.ft3_basic_dev;

query get_version() {
    return "0.0.2";
}

```

If everything works correctly we will not receive any response when running the module `v0_0_1` but we will receive `0_0_2` when running the module `v0_0_2`.

## Set the migration to new module

To migrate to the new module, we need to update the `config.template.xml` file.

We will add a new `<config>` tag inside our `<chain>`, and set it to be enabled at a certain (block) height:

```

<run wipe-db="true">
  <nodes>
    <config src="../../node-config.properties" add-signers="true" />
  </nodes>
  <chains>
    <chain name="YOUR_CHAIN_NAME" iid="0">
      <config height="0">
        <app module="v0_0_1">
          <args module="v0_0_1.lib.ft3.core">
            <!-- ... (your chain configs) -->
          </args>
        </app>
      </config>
      <config height="20">
        <app module="v0_0_2">
          <args module="v0_0_2.lib.ft3.core">
            <!-- ... (your chain configs) -->
          </args>
        </app>
      </config>
    </chain>
  </chains>
</run>

```

In this example, after 20 blocks the chain will change app module to `v0_0_2`.

---

**Important:** Ensure that the block height for the new configuration is safely higher than current height.

Setting a block height that is lower than current height might cause problems to nodes that want to sync by re-applying all the transactions.

---

## Test the chain

Start our chain with

```
./postchain/bin/run-node.sh dev
```

Recall the new query `get_version()` we created earlier. We can test the chain's version with a simple script such as:

```
const pcl = require('postchain-client');
const node_api_url = "http://localhost:YOUR_NODE_PORT";

// default blockchain identifier used for testing
const blockchainRID = YOUR_NODE_BRID;

const rest = pcl.restClient.createRestClient(node_api_url, blockchainRID, 5);
const gtvHash = pcl.gtv.gtvHash;

const gtx = pcl.gtxClient.createClient(
  rest,
  Buffer.from(blockchainRID, 'hex'),
  [] );

const gtv = pcl.gtv;
const gtxU = pcl.gtx;

( async () => {
  try {
    const version = await gtx.query("get_version", {});
  } catch (e) {
    console.log("New version not yet implemented");
  } finally {
    if(version) {
      console.log(`New version running: ${version}`)
    }
  }
} ) ();
```

We can monitor the blockchain by linking it to the [Explorer](#) :

chromia-develop ▾

Search block, transaction or account hash

HEIGHT  
22

AVG BLOCK TIME  
~ 28.7 s

CURRENT LOAD  
~ 0.000 tx/s

Latest Blocks ↻

Height	Block Id	Transactions	Age
22	d265 - e851	0	less than a minute
21	02dc - 65e3	0	1 minute
20	5e5c - bbe2	0	1 minute
19	9b50 - 0cd6	0	2 minutes
18	1945 - c4df	0	2 minutes
17	623c - c1c7	0	3 minutes

VIEW MORE

Latest Transactions ↻

Transaction Id	Signers	Age
----------------	---------	-----

VIEW MORE

At block 20 we will start getting a result back from the chain. It will return something like:

```
POST URL http://localhost:7743/query/
↪A54320E7D063AB94513467806FE800A1B95B26BF65C4F11D30C5D65859E4025C
0.0.2
```

## Update by referencing old configuration

The second approach that can be used is to specify previous compiled configurations in the config tag.

Use this approach if you want cleaner code structure.

**Note:** An example project to demonstrate this approach is available at branch `feature/CCD210-update-chain-tutorial` of `develop-chromia`:

```
git clone https://bitbucket.org/chromawallet/develop-chromia.git
git checkout feature/CCD210-update-chain-tutorial
```

## What does run-node.sh do?

In order to discuss this approach, we must first explain what happened when you execute `./postchain/bin/run-node.sh dev` in the console.

This command does 3 things.

1. First it creates a target folder where the compiled blockchain configuration will be placed, something like:

```
mkdir -p ./postchain/runtime/nodes/dev
```

2. Then it generates the blockchain configuration by compiling the Rell code with the `config.template.xml` file.

If it is a new chain (e.g. started with the `-W` option), the BRID is also generated at this step. Since we are discussing updating a chain here, we don't want the BRID to change, so `-W` option should never be called while updating.

```
./postchain/lib/multigen.sh ./postchain/config/nodes/dev/blockchains/app/config.  
→template.xml -d ./rell/src -o ./postchain/runtime/nodes/dev/
```

It should create the following architecture inside `postchain/runtime/nodes/dev`:

```
postchain/runtime/nodes/dev
├── blockchains
│   └── 0
│       ├── 0.gtv (creates a one file rell module)
│       ├── 0.xml (creates the configuration that will be run, also by using the just_
│       →created 0.gtv)
│       └── brid.txt (just a file where is output the brid of the chain, n.b. it will_
│       →remain constant nevertheless the updates)
├── node-config.properties
└── private.properties
```

3. Finally, it runs the new configuration with

```
./postchain/lib/postchain.sh run-node-auto -d ./postchain/runtime/nodes/dev
```

## Backup old configuration

Compile the current code (by executing `./postchain/bin/run-node.sh dev` or only step 1 and 2 described above).

In the generated configuration folder, copy `runtime/nodes/dev/0.xml` into `postchain/config/nodes/dev/blockchains/app`, and rename it `0.0.1.xml`

## Adding new codes

In the previous approach, we were keeping track from a code point of view of previous versions. This time we will not do that, so we can change the current code directly without copying anything.

Let's just add the query into the `module.rell` file:

```
module;

import lf: .lib.ft3.ft3_basic_dev;

query get_version() {
    return "0.0.2";
}
```

## Set the migration to new module

Update the `config.template.xml` file to config the migration.

Note that unlike the other approach, this time we don't have to change the `<app>` and `<args>` tags:

```
<run wipe-db="true">
  <nodes>
    <config src="../../node-config.properties" add-signers="true" />
  </nodes>
  <chains>
    <chain name="YOUR_CHAIN_NAME" iid="0">
      <config height="0">
        <gtv src="0.0.1.xml"/>
      </config>
      <config height="20">
        <app module="">
          <args module="lib.ft3.core">
            <!-- ... (your chain configs) -->
          </args>
        </app>
      </config>
    </chain>
  </chains>
</run>
```

---

**Important:** Ensure that the block height for the new configuration is safely higher than current height.

Setting a block height that is lower than current height might cause problems to nodes that want to sync by re-applying all the transactions.

---

Compile the new configuration and start the chain:

```
./postchain/bin/run-node.sh dev
```

We can now *Test the chain* as before. If we try to query for `get_version()` after block 20 we will see `0.0.2`.

## 2.6.4 Postchain REST API

Postchain is the blockchain framework that Chromia is built on. In the Chromia ecosystem, Providers run Postchain nodes that form the infrastructure for the network. Sometimes it makes sense to run your own node or set of nodes, such as for testing, development, or even for setting up your own consortium blockchain network.

Postchain nodes expose a REST API that allows for submitting transactions, posting queries, fetching block info, reading node status, etc. You can .



## 2.6.5 GTX

GTX is a protocol that Postchain uses for sending and receiving transactions. GTX is encoded into the base protocol GTV.

It's not recommended to use GTV directly, even though it is possible to construct various Rell types directly into GTV.

```
GtxMessages DEFINITIONS ::= BEGIN

IMPORTS RawGtv FROM GtvMessages;

-- All types are using the same tags as RawGtv to make gtv<->gtx encoding equivalent

-- Gtx operation
-- [ string, [ gtv ] ]
RawGtxOp ::= [5] EXPLICIT SEQUENCE {
    name [2] EXPLICIT UTF8String, -- RawGtv { string } --
    args [5] EXPLICIT SEQUENCE OF RawGtv -- RawGtv { array } --
}

-- Gtx Body
-- [ bytearray, [ gtxOp ], [ bytearray ] ]
RawGtxBody ::= [5] EXPLICIT SEQUENCE {
    blockchainRid [1] EXPLICIT OCTET STRING, -- RawGtv { bytearray } --
    operations [5] EXPLICIT SEQUENCE OF RawGtxOp,
    signers [5] EXPLICIT SEQUENCE OF [1] EXPLICIT OCTET STRING -- RawGtv {
↳bytearray } --
}

-- Gtx
-- [ gtxBody, [ bytearray ] ]
RawGtx ::= [5] EXPLICIT SEQUENCE {
    body RawGtxBody,
    signatures [5] EXPLICIT SEQUENCE OF [1] EXPLICIT OCTET STRING -- RawGtv
↳{ bytearray } --
}

END
```

## GTV

GTV is a general purpose protocol, used to express data in;

- Primitive types (integers, strings, byte arrays, etc)
- Complex types (only Array and Dictionary)

GTV is converted to ASN when it's sent.

```
GtvMessages DEFINITIONS ::= BEGIN

    DictPair ::= SEQUENCE {
        name UTF8String,
        value RawGtv
    }

    RawGtv ::= CHOICE {
        null [0] NULL,
        byteArray [1] OCTET STRING,
```

(continues on next page)

(continued from previous page)

```

    string [2] UTF8String,
    integer [3] INTEGER,
    dict [4] SEQUENCE OF DictPair,
    array [5] SEQUENCE OF RawGtv,
    bigInteger [6] INTEGER
}
END

```

## Type conversions

Below here is a table to visualize the type conversions that happens in the the Rel backend to its corresponding Gtv types when inserting into, and querying the types.

Types	Input	Output	Remark
entity	GtvInteger	GtvInteger	
enum	GtvInteger	GtvInteger	Can be GtvString as input to query
struct	GtvArray	GtvDict	Can be GtvDict as input to query
decimal	GtvString	GtvString	
boolean	GtvInteger	GtvInteger	
rowid	GtvInteger	GtvInteger	
json	GtvString	GtvString	
nullable	GtvNull or type	GtvNull or type	
collection	GtvArray	GtvArray	Both set and list
map	GtvDict	GtvDict	If key is text
map	GtvArray	GtvArray	If key is not text [[k1, v1], [k2, v2], ...]
tuple	GtvDict	GtvDict	If named "(x = 1, y = 2, z = 3)"
tuple	GtvArray	GtvArray	If not named "(1, 2, 3)"

## 2.7 Run.XML

Run.XML format is used to define a run-time configuration of a Rel node. The configuration consists of two key parts:

1. The list of Postchain nodes (the target node is one of those nodes).
2. The list of blockchains, each having an associated configuration(s) and a Rel application.

The format is used:

- By Rel command-line utilities `multirun.sh` and `multigen.sh`.
- By the Eclipse IDE (which internally uses `multirun.sh` to launch Postchain applications).

### 2.7.1 The Format

Example of a Run.XML file:

```

<run wipe-db="true">
  <nodes>
    <config src="config/node-config.properties" add-signers="false" />
  </nodes>

```

(continues on next page)

(continued from previous page)

```

<chains>
  <chain name="user" iid="1">
    <config height="0">
      <app module="user" />
      <gtv path="gtx/reII/moduleArgs/user">
        <dict>
          <entry key="foo"><bytea>
↪ 0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15</bytea></entry>
        </dict>
      </gtv>
    </config>
    <config height="1000">
      <app module="user_1000">
        <args module="user_1000">
          <arg key="foo"><bytea>
↪ 0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15</bytea></arg>
        </args>
      </app>
      <gtv path="path" src="config/template.xml"/>
    </config>
  </chain>
  <chain name="city" iid="2">
    <config height="0" add-dependencies="false">
      <app module="city" />
      <gtv path="signers">
        <array>
          <bytea>
↪ 0350fe40766bc0ce8d08b3f5b810e49a8352fdd458606bd5fafe5acdcdc8ff3f57</bytea>
        </array>
      </gtv>
      <dependencies>
        <dependency name="user" chain="user" />
      </dependencies>
    </config>
    <include src="config/city-include-1.xml"/>
    <include src="config/city-include-2.xml" root="false"/>
  </chain>
</chains>
</run>
    
```

Top-level elements are:

- nodes - defines Postchain nodes
- chains - defines blockchains

## Nodes

Node configuration is provided in a standard Postchain node-config.properties format.

Specifying a path to an existing node-config.properties file (path is relative to the Run.XML file):

```

<nodes>
  <config src="config/node-config.properties" add-signers="false" />
</nodes>
    
```

Specifying node configuration properties directly, as text:

```
<nodes>
  <config add-signers="false">
    database.driverclass=org.postgresql.Driver
    database.url=jdbc:postgresql://localhost/postchain
    database.username=postchain
    database.password=postchain
    database.schema=test_app

    activechainids=1

    api.port=7740
    api.basepath=

    node.0.id=node0
    node.0.host=127.0.0.1
    node.0.port=9870
    node.0.
    ↳ pubkey=0350fe40766bc0ce8d08b3f5b810e49a8352fdd458606bd5fafe5acdcdc8ff3f57

    messaging.
    ↳ privkey=3132333435363738393031323334353637383930313233343536373839303131
    messaging.
    ↳ pubkey=0350fe40766bc0ce8d08b3f5b810e49a8352fdd458606bd5fafe5acdcdc8ff3f57
  </config>
</nodes>
```

## Chains

A chain element can have multiple `config` elements and a `dependencies` element inside.

A single chain may have specific configurations assigned to specific block heights.

```
<config height="0" add-dependencies="false">
  <app module="city" />
  <gtv path="signers">
    <array>
      <bytea>0350fe40766bc0ce8d08b3f5b810e49a8352fdd458606bd5fafe5acdcdc8ff3f57
    ↳ </bytea>
    </array>
  </gtv>
</config>
```

An `app` element specifies a Rel application used by the chain. Attribute `module` is the name of the main module of the app. The source code of the main module and all modules it imports will be injected into the generated blockchain XML configuration.

Elements `gtv` are used to inject GTXML fragments directly into the generated Postchain blockchain XML configuration. Attribute `path` specifies a dictionary path for the fragment (default is root). For example, the fragment

```
<gtv path="gtx/rell/moduleArgs/user">
  <dict>
    <entry key="foo"><bytea>
    ↳ 0373599a61cc6b3bc02a78c34313e1737ae9cfd56b9bb24360b437d469efdf3b15</bytea></entry>
  </dict>
</gtv>
```

will produce a blockchain XML:

```

<dict>
  <entry key="gtx">
    <dict>
      <entry key="rell">
        <dict>
          <entry key="moduleArgs">
            <dict>
              <entry key="user">
                <dict>
                  <entry key="foo">
                    <bytea>
→ 0373599A61CC6B3BC02A78C34313E1737AE9CFD56B9BB24360B437D469EFDF3B15</bytea>
                    </entry>
                  </dict>
                </entry>
              </dict>
            </entry>
          </dict>
        </entry>
      </dict>
    </entry>
  </dict>
</dict>

```

GTXML contents to be injected shall be either specified as a nested element of a `gtx` element, or placed in an XML file referenced via the `src` attribute.

### Included files

Other XML files can be included anywhere in a Run.XML using `include` tag. Included files may include other XML files as well.

Including a file with its root element replacing the `include` element:

```
<include src="config/city-include-1.xml"/>
```

Including a file without its root element, the `include` is replaced by the child elements of the root element of the file:

```
<include src="config/city-include-2.xml" root="false"/>
```

## 2.7.2 Utilities

These utilities are a part of the [Rell](#) language.

### multirun.sh

Runs an application described by a Run.XML configuration.

```

Usage: RellRunConfigLaunch [-d=SOURCE_DIR] RUN_CONFIG
Launch a run config
    RUN_CONFIG    Run config file
    -d, --source-dir=SOURCE_DIR
                  Rell source code directory (default: current directory)

```

## multigen.sh

Creates a Postchain blockchain XML configuration from a Run.XML configuration.

```
Usage: RelRunConfigGen [--dry-run] [-d=SOURCE_DIR] [-o=OUTPUT_DIR] RUN_CONFIG
Generate blockchain config from a run config
    RUN_CONFIG    Run config file
    --dry-run     Do not create files
    -d, --source-dir=SOURCE_DIR
                  Rel source code directory (default: current directory)
    -o, --output-dir=OUTPUT_DIR
                  Output directory
```

Example of a generated directory tree:

```
out/
├── blockchains
│   ├── 1
│   │   ├── 0.gtv
│   │   ├── 0.xml
│   │   ├── 1000.gtv
│   │   ├── 1000.xml
│   │   └── brid.txt
│   └── 2
│       ├── 0.gtv
│       ├── 0.xml
│       ├── 1000.gtv
│       ├── 1000.xml
│       ├── 2000.gtv
│       ├── 2000.xml
│       ├── 3000.gtv
│       ├── 3000.xml
│       └── brid.txt
├── node-config.properties
└── private.properties
```

## 2.8 Testing module

To write unit tests for Rel code, use test modules. A test module is defined using the `@test` annotation:

```
@test module;

function test_foo() {
    assert_equals(2 + 2, 4);
}

function test_bar() {
    assert_equals(2 + 2, 5);
}
```

All functions in a test module that start with `test_` (and a function called exactly `test`) are test functions and will be executed when the test module is run.

Tests are executed using a postchain node with the following signer key:

```
val signer_privkey = x  
↳ "4242424242424242424242424242424242424242424242424242424242424242";  
val signer_pubkey = x  
↳ "0324653EAC434488002CC06BBFB7F10FE18991E35F9FE4302DBEA6D2353DC0AB1C";
```

To run a test module, use the command-line interpreter:

```
rell.sh -d my_src_directory --test my_test_module
```

or right-click on the run.xml file and run as **Repl Unit Test**

Each test function will be executed independently of others, and a summary will be printed in the end:

```
TEST RESULTS:

my_test_module:test_foo OK
my_test_module:test_bar FAILED

SUMMARY: 1 FAILED / 1 PASSED / 2 TOTAL

***** FAILED *****
```

### 2.8.1 Transactions in test module

Instead of writing tests in a frontend, we write the transactions in rell like this:

```
rell.test.block - a block, contains a list of transactions
```

rell.test.tx - a transaction, has a list of operations and a list of signers

`rell.test.op` - an operation call, which is a (mount) name and a list of arguments (each argument is a gtv)

Keys used for signing transactions can also be created like this:

```

rell.test.keypairs.{bob, alice, trudy}: rell.test.keypair - test key-
pairs rell.test.privkeys.{bob, alice, trudy}: byte_array - same as
rell.test.keypairs.X.priv rell.test.pubkeys.{bob, alice, trudy}: byte_array -
same as rell.test.keypairs.X.pub

```

Example of a operation signed with the alice keypair:

```
rell.test.tx().op(main.exampleOp(parameter1,parameter2)).sign(rell.test.keypairs.alice).run();
```

And if if nop is necessary for making a transaction unique, one can use `ℓell.test.nop()` to implement it.

```
rell.test.nop(x: integer): rell.test.op      rell.test.nop(x: text):
rell.test.op rell.test.nop(x: byte_array): rell.test.op
```

Creates a “nop” operation with a specific argument value.

### 2.8.2 Building and running a block

```
operation foo(x: integer) { ... }
operation bar(s: text) { ... }

...

val tx1 = rell.test.tx()
    .op(foo(123))           // operation call returns rell.test.op
    .op(bar('ABC'))        // now the transaction has two operations
    .sign(rell.test.keypairs.bob) // signing with the "Bob" test keypair
    ;

val tx2 = rell.test.tx()
    .op(bar('XYZ'))
    .sign(rell.test.keypairs.bob)
    .sign(rell.test.keypairs.alice) // tx2 is signed with both "Bob" and
    ↪ "Alice" keypairs
    ;

rell.test.block()
    .tx(tx1)
    .tx(tx2)
    .run()                 // execute the block consisting of two_
    ↪ transactions: tx1 and tx2
    ;
```

If we our module has the “\_test” suffix it will become a test module for the module that bears the same name without the suffix. For example, if our modules name is program, the test module would be called program\_test.

## 2.8.3 Production and test modules

Production module (file data.rell):

```
module;

entity user {
    name;
}

operation add_user(name) {
    create user(name);
}
```

Test module (file data\_test.rell):

```
@test module;
import data;

function test_add_user() {
    assert_equals(data.user@*{}(.name), list<text>());

    val tx = rell.test.tx(data.add_user('Bob'));
    assert_equals(data.user@*{}(.name), list<text>());
}
```

(continues on next page)



(continued from previous page)

```
tx.run();
assert_equals(data.user@*{}(.name), ['Bob']);
}
```

## Functions of `rell.test.block`

`rell.test.block()` - create an empty block builder  
`rell.test.block(tx: rell.test.tx, ...)` - create a block builder with some transaction(s)  
`rell.test.block(txs: list<rell.test.tx>)` - same  
`rell.test.block(op: rell.test.op, ...)` - create a block builder with one transaction with some operation(s)  
`rell.test.block(ops: list<rell.test.op>)` - same

`.tx(tx: rell.test.tx, ...)` - add some transaction(s) to the block  
`.tx(txs: list<rell.test.tx>)` - same  
`.tx(op: rell.test.op, ...)` - add one transaction with some operation(s) to the block  
`.tx(ops: list<rell.test.op>)` - same

`.copy()`: `rell.test.block` - returns a copy of this block builder object  
`.run()` - run the block  
`.run_must_fail()` - same as `.run()`, but throws exception on success, not on failure

## Functions of `rell.test.tx`:

`rell.test.tx()` - create an empty transaction builder  
`rell.test.tx(op: rell.test.op, ...)` - create a transaction builder with some operation(s)  
`rell.test.tx(ops: list<rell.test.op>)` - same

`.op(op: rell.test.op, ...)` - add some operation(s) to this transaction builder  
`.op(ops: list<rell.test.op>)` - same

`.nop()` - same as `.op(rell.test.nop())`  
`.nop(x: integer)` - same as `.op(rell.test.nop(x))`  
`.nop(x: text)` - same  
`.nop(x: byte_array)` - same

`.sign(keypair: rell.test.keypair, ...)` - add some signer keypair(s)  
`.sign(keypairs: list<rell.test.keypair>)` - same  
`.sign(privkey: byte_array, ...)` - add some signer private key(s) (a private key must be 32 bytes)  
`.sign(privkeys: list<byte_arrays>)` - same

`.copy()`: `rell.test.tx` - returns a copy of this transaction builder object

`.run()` - runs a block containing this single transaction  
`.run_must_fail()` - same as `.run()`, but throws exception on success, not on failure

### Functions of `rell.test.op`:

`rell.test.op(name: text, arg: gtv, ...)` - creates an operation call object with a given name and arguments  
`rell.test.op(name: text, args: list<gtv>)` - same

`.tx(): rell.test.tx` - creates a transaction builder object containing this operation  
`.sign(...): rell.test.tx` - equivalent of `.tx().sign(...)`  
`.run()` - equivalent of `.tx().run()`  
`.run_must_fail()` - equivalent of `.tx().run_must_fail()`

Functions `assert_*` for unit tests

### Other functions:

`assert_equals(actual: T, expected: T)` - fail (throw an exception) if two values are not equal  
`assert_not_equals(actual: T, expected: T)` - fail if the values are equal

`assert_true(actual: boolean)` - assert that the value is “true”  
`assert_false(actual: boolean)` - assert that the value is “false”

`assert_null(actual: T?)` - assert that the value is null  
`assert_not_null(actual: T?)` - assert that the value is not null

`assert_lt(actual: T, expected: T)` - assert less than (`actual < expected`)  
`assert_gt(actual: T, expected: T)` - assert greater than (`actual > expected`)  
`assert_le(actual: T, expected: T)` - assert less or equal (`actual <= expected`)  
`assert_ge(actual: T, expected: T)` - assert greater or equal (`actual >= expected`)

`assert_gt_lt(actual: T, min: T, max: T)` - assert (`actual > min`) and (`actual < max`)  
`assert_gt_le(actual: T, min: T, max: T)` - assert (`actual > min`) and (`actual <= max`)  
`assert_ge_lt(actual: T, min: T, max: T)` - assert (`actual >= min`) and (`actual < max`)  
`assert_ge_le(actual: T, min: T, max: T)` - assert (`actual >= min`) and (`actual <= max`)

Same functions are also available in the `rell.test` namespace.

## Running unit tests via run.xml

With the help of run.xml we can run tests with module arguments included (constants we define in the run xml file). To define a test in xml, we will use the `test` tag like this:

```
<run>
  <nodes>
    <config src="node-config.properties" />
    <test-config src="node-config-test.properties" />
  </nodes>

  <chains>
    <chain name="foo" iid="1">
      <config height="0">
        <app module="foo.app" />
      </config>
      <test module="foo.tests" />
    </chain>

    <test module="lib.tests" />
  </chains>
</run>
```

We will run the test like this in the terminal: `./multirun.sh -d rell/src --test rell/config/run.xml`

## Example of a Test Module

Here is an example of a test module implemented for the Chroma-Chat example that can be found in the Example Projects section.

```
@test module;
import main;

function test_init() {

  assert_equals((main.user@*{}(.username)).size(), 0);
  assert_equals((main.balance@*{}(.user)).size(), 0);
  rell.test.tx().op(main.init(rell.test.pubkeys.alice)).run();
  assert_equals(main.user@*{}(.username).size(), 1);
  assert_equals(main.balance@{.user == main.user@{.pubkey == rell.test.pubkeys.
↪alice}}(.amount), 1000000);
}

function test_register_user() {

  rell.test.tx().op(main.init(rell.test.pubkeys.alice)).run();
  assert_equals(main.user@*{}(.username).size(), 1);
  rell.test.tx().op(main.register_user(rell.test.pubkeys.alice, rell.test.pubkeys.
↪bob, "bob", 100)).sign(rell.test.keypairs.alice).run();
  assert_equals(main.user@*{}(.username).size(), 2);
  assert_equals(main.user@{.pubkey == rell.test.pubkeys.bob}(.username), "bob");
  assert_equals(main.balance@{.user == main.user@{.pubkey == rell.test.pubkeys.bob}}
↪(.amount), 100);
}
```

(continues on next page)

(continued from previous page)

```
function test_blocks() {
    val tx1 = rell.test.tx().op(main.init(rell.test.pubkeys.alice));
    val tx2 = rell.test.tx().op(main.register_user(rell.test.pubkeys.alice, rell.test.
    ↪pubkeys.bob, "bob", 100)).sign(rell.test.keypairs.alice);
    val tx3 = rell.test.tx().op(main.create_channel(rell.test.pubkeys.alice, "channel 1
    ↪")).sign(rell.test.keypairs.alice);
    rell.test.block().tx(tx1).tx(tx2).tx(tx3).run();
    val tx4 = rell.test.tx().op(main.add_channel_member(rell.test.pubkeys.alice,
    ↪"channel 1", "bob")).sign(rell.test.keypairs.alice);
    rell.test.block().tx(tx4).run();
}
```

## Running tests in docker

If one is running their program from a docker instance, testing will be slightly different. Tests will run through a script that will run the tests specified in the test module.

---

**Note:** For new projects, [this project template](#) can be built upon that has existing docker files needed and the script that runs tests.

---

Implementing docker test functionality into an already existing project means pasting [this](#) script manually into your project folder. Alongside the script, a docker compose file that specifies from where and how the tests will run will also be needed.

The docker-compose.yml file will look something like this:

```
version: '3.3'
services:
  test_postgres:
    image: postgres:14.1-alpine
    container_name: test_postgres
    restart: always
    environment:
      POSTGRES_DB: postchain
      POSTGRES_USER: postchain
      POSTGRES_PASSWORD: postchain
  test_blockchain:
    image: registry.gitlab.com/chromaway/postchain-distribution/chromaway/postchain-
    ↪test-dapp:3.5.0
    container_name: test_blockchain
    command:
      - ${COMMAND}
    depends_on:
      - test_postgres
    volumes:
      - ./rell:/opt/chromaway/rell
    environment:
      POSTCHAIN_DB_URL: jdbc:postgresql://test_postgres/postchain
      CHAIN_CONF: /opt/chromaway/rell/config/run.xml
      NODE_CONF: /opt/chromaway/rell/config/node-config.properties
```

to run your tests, simply run the script by writing the following inside the folder rell-tests.js is inside:

```
node rell-tests.js
```

## 2.9 Upgrading to Rell 0.10.x

There are two kinds of breaking changes in Rell 0.10.x:

### 1. Rell Language:

- Module System; `include` is deprecated and will not work.
- Mount names: mapping of entities and objects to SQL tables changed.
- `class` and `record` renamed to `entity` and `struct`, the code using old keywords will not compile.
- Previously deprecated library functions are now unavailable; the code using them will not compile.

### 2. Configuration and tools:

- Postchain `blockchain.xml`: now needs a list of modules instead of the main file name; module arguments are per-module.
- Run.XML format: specifying module instead of main file; module arguments are per-module.
- Command-line tools: accept a source directory path and main module name combination instead of the main `.rell` file path.

### 2.9.1 Step-by-step upgrade

1. Read about the *Module System*.
2. Read about *mount names*.
3. Use the `migrate-v0.10.sh` tool to rename `class`, `record` and deprecated function names (see below).
4. Manually update the source code to use the Module System instead of `include`.
5. Use `@mount` annotation to set correct mount names to entities, objects, operations and queries (recommended to apply `@mount` to entire modules or namespaces, not to individual definitions).
6. Update configuration files, if necessary (see the details below).
7. The Web IDE users the root module as the main module, so make sure you have it and import all required modules there.

### 2.9.2 Details

#### migrate-v0.10.sh tool

The tool can be found in the `postchain-node` directory of a Rell distribution. It renames `class`, `record` and most of deprecated functions, e. g. `requireNotEmpty()` -> `require_not_empty`.

```
Usage: migrator [--dry-run] DIRECTORY
Replaces deprecated keywords and names in all .rell files in the directory_
↪ (recursively)
    DIRECTORY    Directory
    --dry-run    Do not modify files, only print replace counts
```

Specify a Rel source directory as an argument, and the tool will do renaming in all .rel files in that directory and its subdirectories.

**NOTE.** UTF-8 encoding is always used by the tool; if files use a different encoding, some characters may be broken. It is recommended to not run the tool if there are uncommitted changes in the directory. After running it, review the changes it made.

## blockchain.xml

New blockchain.xml Rel configuration looks like this (only changed parts shown):

```
<dict>
  <entry key="gtx">
    <dict>
      <entry key="rell">
        <dict>
          <entry key="moduleArgs">
            <dict>
              <entry key="app.foo">
                <dict>
                  <entry key="message">
                    <string>Some common message...</string>
                  </entry>
                </dict>
              </entry>
              <entry key="app.bar">
                <dict>
                  <entry key="x">
                    <int>123456</int>
                  </entry>
                  <entry key="y">
                    <string>Hello!</string>
                  </entry>
                </dict>
              </entry>
            </dict>
          </entry>
          <entry key="modules">
            <array>
              <string>app.foo</string>
              <string>app.bar</string>
            </array>
          </entry>
        </dict>
      </entry>
    </dict>
  </entry>
</dict>
```

What was changed:

- gtx.rell.moduleArgs is now a dictionary, specifying module\_args for multiple modules (in older versions there was only one module\_args for a Rel application, now there can be one module\_args per module).
- gtx.rell.modules is an array of module names

## run.xml

An example of a new `run.xml` file:

```
<run wipe-db="true">
  <nodes>
    <config src="node-config.properties" add-signers="false" />
  </nodes>
  <chains>
    <chain name="test" iid="1" brid=
↪ "01234567abcdef01234567abcdef01234567abcdef01234567abcdef01234567">
      <config height="0">
        <app module="app.main">
          <args module="app.bar">
            <arg key="x"><int>123456</int></arg>
            <arg key="y"><string>Hello!</string></arg>
          </args>
          <args module="app.foo">
            <arg key="message"><string>Some common message...</string></
↪ arg>
          </args>
        </app>
      </config>
    </chain>
  </chains>
</run>
```

What was changed:

- `module` tag replaced by `app`, which has `module` attribute
- there can be multiple `args` elements, each must have a `module` attribute