
Rel Documentation

Release v0.10.0

ChromaWay AB

Dec 16, 2019

Contents

1	Rell language	3
2	Chromia	5
2.1	Get Started with Web IDE	5
2.2	Rell Basics	10
2.3	Example Projects	19
2.4	Language Features	30
2.5	Advanced Topics	65
2.6	Upgrading to Rell 0.10	93
2.7	Eclipse IDE	96
2.8	Run.XML	129

This section discuss Reli and its position within the Chromia platform.

If you are eager to get started with Reli, you can safely skip straight to the *Quick Start* section.

For code references, visit *Language Features*.

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

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

Rell 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:

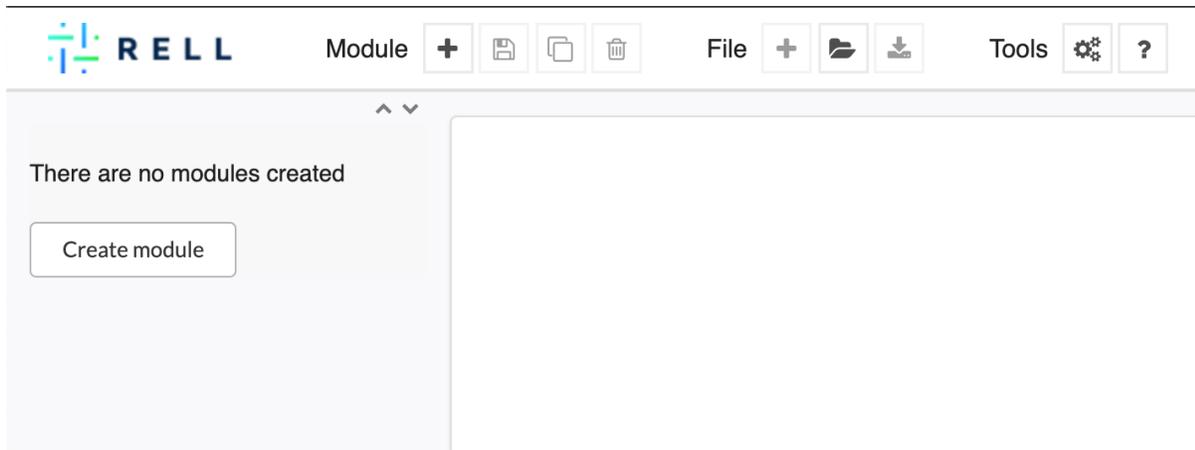
- Allow dapps to scale to millions of users.
- Improve the user experience of dapps to achieve parity with centralized applications.
- Allow developers to build secure applications with using familiar paradigms.

2.1 Get Started with Web IDE

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

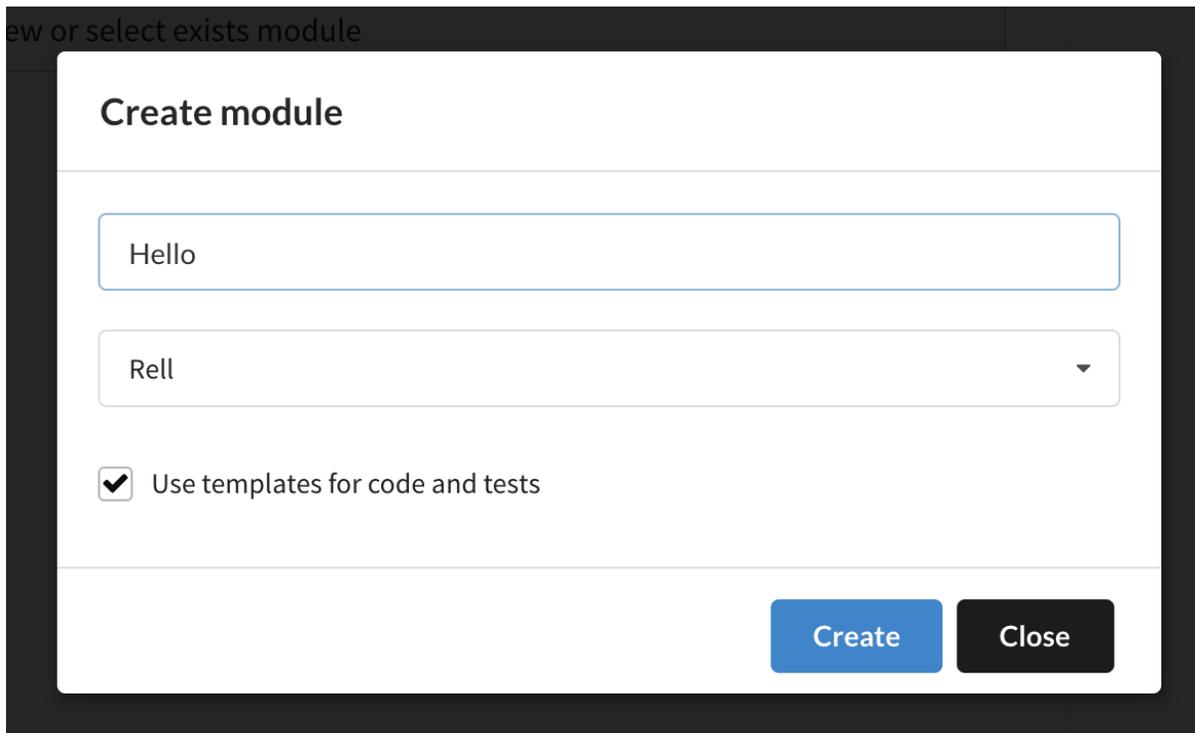
Upon entering, you will see an interface similar to the image below.

Click **Create Module** button:

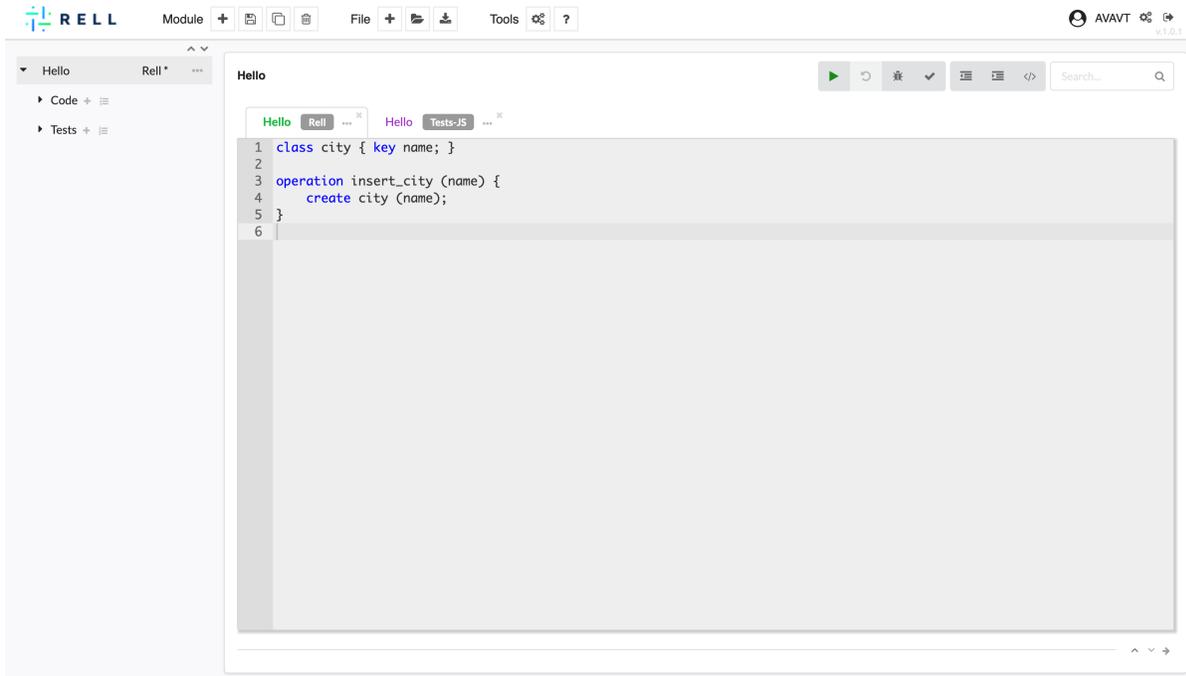


This will open a modal element where you can specify the name of the Module and the Language used (in this example: Rell).

For convenience you can include template and test code.



Click the blue “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 Config ReII version

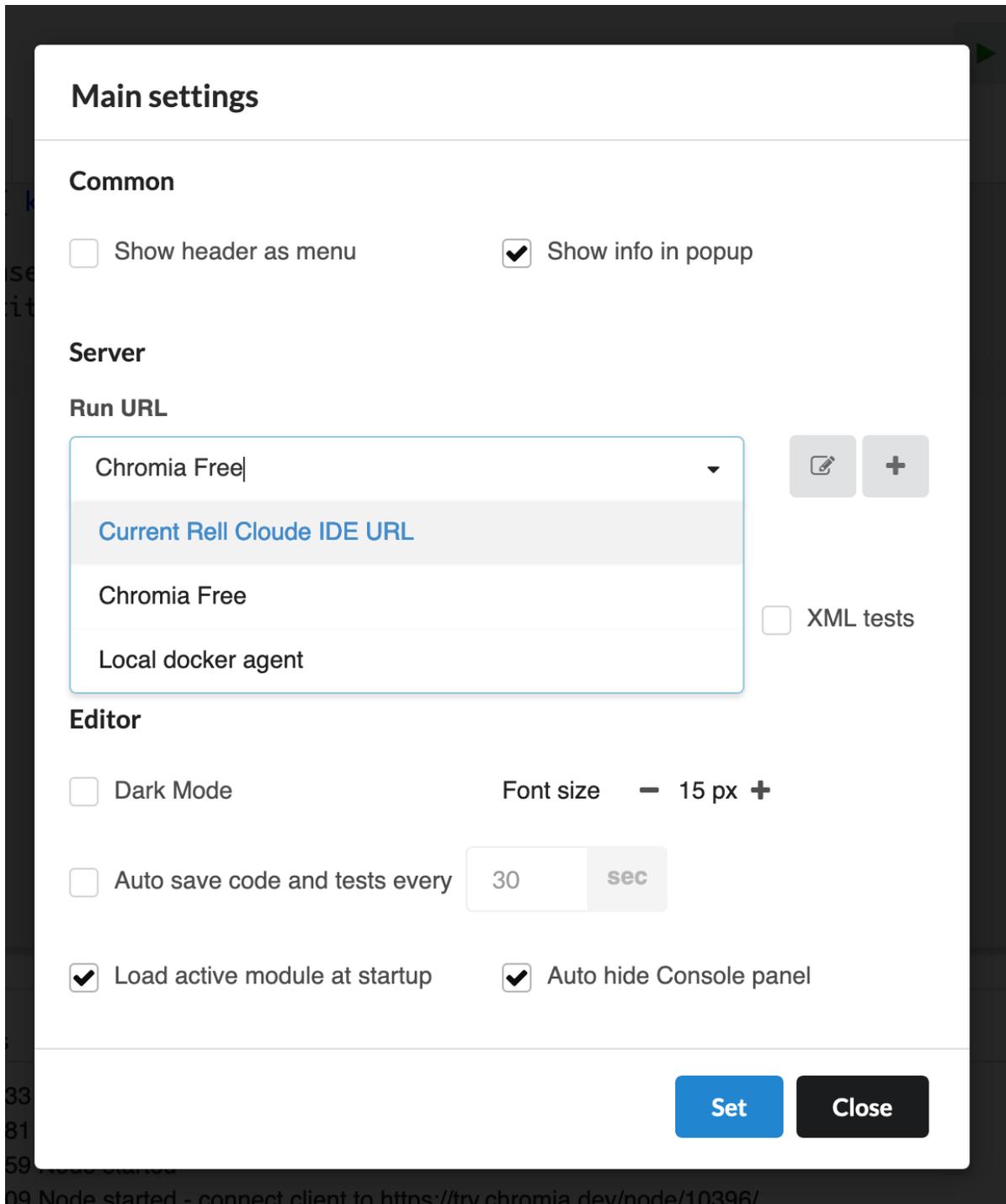
Warning: At current time, WebIDE use reII version 0.9.1 by default. If you Start the node now, you will see errors printed in the console because of syntax mismatch.

We will need to config our module to use version 0.10.x.

On the top menu, click the Main Settings button:



A popup will appear to config for current module. Expand the *Server -> Run URL* dropdown and choose **Current ReII Cloud IDE Url**:



Click the blue “Set” button. Our module will now use latest ReII version.

2.1.2 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.

The screenshot shows a web browser with two tabs: 'Hello Rel' and 'Hello Tests-JS'. The 'Tests-JS' tab is active, displaying a code editor with the following JavaScript code:

```

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

```

Below the code editor is a test runner interface with tabs for 'Session', 'JS tests', 'XML tests', 'Log*', and 'Console'. The 'JS tests' tab is selected, showing a green checkmark and the text 'Tests passed' with a refresh icon. Below this, it says 'Output: null'.

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

2.1.3 Where to go next?

Next step is to learn about what those Rel code actually mean 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 Rel Basics

In this chapter we discuss fundamental concepts of Rel language.

- The *Main Concepts* section guide you through the concepts needed to create node backend with Rell.
- While *Client Side* discuss how to work with such 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 a entity definition automatically creates the storage (e.g. a table) necessary to persist objects of a entity. As you might expect, Rell's entity definition includes a list of attributes:

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

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. Rel 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 Rel. 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, Rel 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.

Relational operator basics

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.

How do we find that object now?

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

`@` 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.

Simple operation

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

```
operation register_channel (user_pubkey: pubkey, channel_name: name) {
    require( is_signer(user_pubkey) );
    create channel (
```

(continues on next page)

(continued from previous page)

```

        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( 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 Rel – 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 operation will fail. We do not need to test for that explicitly as `@` operator will do this job.

Rel 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 a 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};
}

```

(continues on next page)

(continued from previous page)

```
operation change_channel_name(signer: pubkey, old_channel_name: name, new_channel_
↳name: name) {
  require(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 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 a 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 later case, conceptually we are dealing with a Cartesian product, which is a set of all possible combinations. But, in typically *WHERE* part will then provide a condition which defines a correspondence between objects of difference 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 Rell 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 m.timestamp) limit 25
```

Here we sorted results by timestamp in a descending order using `-sort` (minus prefix means descending) and limited the number of returned rows.

Composite indices

We can also only 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 Rell. But in a decentralized system resources are typically precious, thus it is important for Rell programmers to understand the query behavior and use indices efficiently.

2.2.2 Client Side

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 Rell.

Try the example code

First of all, we need to add a query to Rell 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, submit to a node. And once transaction is added to a block, client will perform a query.

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

Install the client

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);
```

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 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 it is created is possible call as many operations as you want.

```
tx.addOperation('insert_city', "Tel Aviv");
tx.addOperation('insert_city', "Stockholm");
/* 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 Rell 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 Rell. In the next section, we have prepared some examples of how to implement other functionalities in Rell and Chromia.

2.3 Example Projects

2.3.1 Chroma Chat

In this section we will write the code for a public chat.

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 (which is specified in the name of the channel, e.g. “showerthoughts”, where you can send messages with the thoughts you had under 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 As said, user is solely identified by their public key

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 an 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: one 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

Init

To initialize the module, we need to have at least one registered user.

We don't want the user to call this function once the admin is set (i.e. we don't want users to change the admin). To prevent such event, we create an operation called `init` which verified that no users are registered and, in case of positive response, creates a new 'admin' user.

```
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).

The interesting point is `require((user@*{} limit 1).size() == 0);`. Here we retrieve a lists of users with a limit of 1: we get the first user in the table. If there is no user, it will return an empty list. Indeed we check its `size()` and if it's 0 we can proceed in running the operation since there are no users registered.

In the third and fourth line the an user with usernam "admin" is created and 1000000 tokens are given to her.

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( 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( 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( 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 (`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( 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 module
- Enter the above code in the code section (You can copy the full code from [here](#)).
- Click on Start Node (The green “Play” icon)

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
```

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

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

(continued from previous page)

```

    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 extends this sample in anyway you like, for example adding an user interace, or maybe add a “transfer” operation to send tokens to another user?

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

2.3.2 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 Rell. There are roughly two different implementation strategies:

- Account-based tokens which maintain an updateable 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:

```

class balance {
    key pubkey;
    mutable amount: integer;
}

operation transfer(from_pubkey: pubkey, to_pubkey: pubkey, xfer_amount: integer) {
    require( 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 updateable 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 Reli 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). Reli-based blockchains can just use built-in indexing to keep track of payment history. For example, by using the additional `payment` class. To make things more efficient, we also wrap `pubkey` into `user` class, thus getting:

```

class user { key pubkey; }

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

class payment {
    index from_user: user;
    index to_user: user;
    amount: integer;
    timestamp;
}

operation transfer(from_pubkey: pubkey, to_pubkey: pubkey, xfer_amount: integer) {
    require( 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,

```

(continues on next page)

(continued from previous page)

```

        timestamp=op_context.last_block_time);
    }

```

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.)

Note: In a future version of Rell it will be possible to timestamp objects automatically using the `log` annotation, with the added benefit that they are then linked to the corresponding transaction and block.

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

```

class asset { key asset_code; }

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

```

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

2.3.3 UTXO-based token system

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

We first define an unspent transaction output structure:

```

class 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(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 that we perform checks as we go. This is OK because Rell is transactional: if a requirement fails or an error is generated, the whole operation (in fact, the whole transaction) is rolled back. Rell 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 Rell (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 Rell programmer can work with: he is allowed to do destructive updates and delete entries. These operations do not affect the raw blockchain.

2.4 Language Features

2.4.1 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*
- *Global Functions*
- *Require function*

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 in a decimal number (131072 + 20)

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

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

`decimal.MIN_VALUE`: `decimal` = the smallest nonzero absolute value that can be accurately stored in a decimal (1E-20)

`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`. Exponential notation is allowed. Rounds the number to 20 decimal places, if necessary. Throws an exception if the number is out of range or not a valid number.

`.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 (5.99 becomes 5, -5.99 becomes -5), throws an exception if the resulting value is out of range

`.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`

`.size()`: `integer`

`.compare_to(text)`: `integer` - as in Java

`.starts_with(text)`: `boolean`

`.ends_with(text)`: `boolean`

`.contains(text)`: `boolean` - true if contains the given substring
`.index_of(text, start: integer = 0)`: `integer` - returns -1 if substring is not found (as in Java)
`.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)`
`.upper_case()`: `text`
`.lower_case()`: `text`
`.split(text)`: `list<text>` - strictly split by a separator (not a regular expression)
`.trim()`: `text` - remove leading and trailing whitespace
`.matches(text)`: `boolean` - true if 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>'`

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'`, throws an exception if the string is not a valid HEX sequence

`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, throws an exception if the string is invalid

`byte_array.from_list(list<integer>)`: `byte_array` - creates a `byte_array` from a list; values must be 0 - 255, otherwise an exception is thrown

`.empty()`: `boolean`

`.size()`: `integer`

`.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`

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

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

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 exist
```

- 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`

`.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

Collection types are:

- `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.

Following types are mutable:

- 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>();

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

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

`list<T>`

Ordered collection type. Accept duplication.

```
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>` can be used only with following types T:

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

- struct
- tuple

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, *>)

Operations available for all virtual types:

- 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(); // type of "rec" is "virtual<rec>", not "rec"
        // type of "full" is "rec", fails if the_
    }
}

```

↳ 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?).
-

Global Functions

abs(integer): integer - absolute value
 abs(decimal): decimal

exists(T?): boolean - returns true if the argument is null and false otherwise

is_signer(byte_array): boolean - returns true if a byte array is in the list of signers of current operation

log(...) - print a message to the log (same usage as print)

max(integer, integer): integer - maximum of two values
 max(decimal, decimal): decimal

min(integer, integer): integer - minimum of two values
 min(decimal, decimal): decimal

print(...) - print a message to STDOUT:

- print() - prints an empty line
- print('Hello', 123) - prints "Hello 123"

verify_signature(message: byte_array, pubkey: pubkey, signature: byte_array): 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

For checking a boolean condition:

require(boolean[, text]) - throws an exception if the argument is false

For 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

For checking for an empty collection:

`require_not_empty(list<T>[, text])`: `list<T>` - throws an exception if the argument is an empty collection, otherwise returns the argument

`require_not_empty(set<T>[, text])`: `set<T>` - same as the previous

`require_not_empty(map<K,V>[, text])`: `map<K,V>` - same as the previous

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
```

2.4.2 Module definitions

Table of Contents

- *Module definitions*
 - *Entity*
 - * *Keys and Indices*
 - * *Entity annotations*
 - *Object*
 - *Struct*
 - *Enum*
 - *Query*
 - *Operation*
 - *Function*
 - *Namespace*
 - *External*
 - * *Transactions and blocks*
 - *Mount names*

Entity

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

(continues on next page)

(continued from previous page)

```

    key first_name, last_name;
}

```

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;
    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.

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');
```

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
```

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;
}
```

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

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

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 {}
}
```

External

External blocks are used to access entities defined in other blockchains:

```
external 'foo' {
  @log entity user {
    name;
  }
}

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

In this example, 'foo' is the name of an external blockchain. To be used in an external block, 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`.

Other features:

- External entities must be annotated with the `@log` annotation. This implies that those entity cannot have mutable attributes.
- Values of external entities cannot be created or deleted.
- Only entities and namespaces are allowed inside of an external block.
- Can have only one external block for a specific blockchain name.
- 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.

Transactions and blocks

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

```
namespace foo {
  external 'foo' {
    entity transaction;
    entity block;
  }
}

function get_foo_transactions(): list<foo.transaction> = foo.transaction @* {};
```

- External block must be put in a namespace in order to prevent name conflict, since entities `transaction` and `block` are already defined in the top-level scope (they represent transactions and blocks of the active blockchain).
- Namespace name can be arbitrary.
- 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).

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`.

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.

In addition, it can be specified for a namespace:

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

or a module:

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

entity user {}
```

In both cases, the mount name of `user` is `foo.bar.user`.

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"
```

2.4.3 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

2.4.4 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 -> {
        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;
}

```

2.4.5 Database Operations

At-Operator

Simplest form:

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

General syntax:

```
<from> <cardinality> { <where> } [<what>] [limit N]
```

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

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

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

Simple example:

`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

Specifying names of result tuple fields:

`user @* {} (x = .company.name, y = .company.address, z = .year_of_birth)` - returns a tuple with named fields (`x, y, z`)

Sorting:

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

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

Field names can be combined with sorting:

`user @* {} (sort x = .last_name, -sort y = .year_of_birth)`

When field names are not specified explicitly, they can be deduced implicitly by 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 prevent implicit field name creation, specify `=` before the expression (i. e. use an “empty” field name):

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

Tail part

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;
```

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 } ( ... )
```

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;
```

2.4.6 System entities

```
entity block {
  block_height: integer;
  block_rid: byte_array;
  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 those entities in code.

chain_context

chain_context.args: module_args - module arguments specified in raw_config under path gtx. rell.moduleArgs.<module name>. 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 {
  s: text;
  n: integer;
}
```

Corresponding module configuration:

```
{
  "gtx": {
    "rell": {
      "moduleArgs": {
        "module_name": {
          "s": "Hello",
          "n": 123
        }
      }
    }
  }
}
```

Code that reads module_args:

```
function f() {
  print(chain_context.args.s);
  print(chain_context.args.n);
}
```

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 of 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.transaction: transaction - the transaction currently being built.

2.4.7 Miscellaneous

Comments

Single-line comment:

```
print("Hello"); // Some comment
```

Multiline comment:

```
print("Hello"/*, "World"*/);
/*
```

(continues on next page)

(continued from previous page)

```
print("Bye");
*/
```

2.5 Advanced Topics

2.5.1 Modules in an Application

Rel application consists of modules. A module is either a single `.rell` file or a directory with one or multiple `.rell` files.

A single-file Rel 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 Rel source directory tree:



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 ReII 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`

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 ReII 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.

2.5.2 Chromia Vault

Short Overview

Chromia Vault is a wallet in nature that supports both same-chain and cross-chain 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.

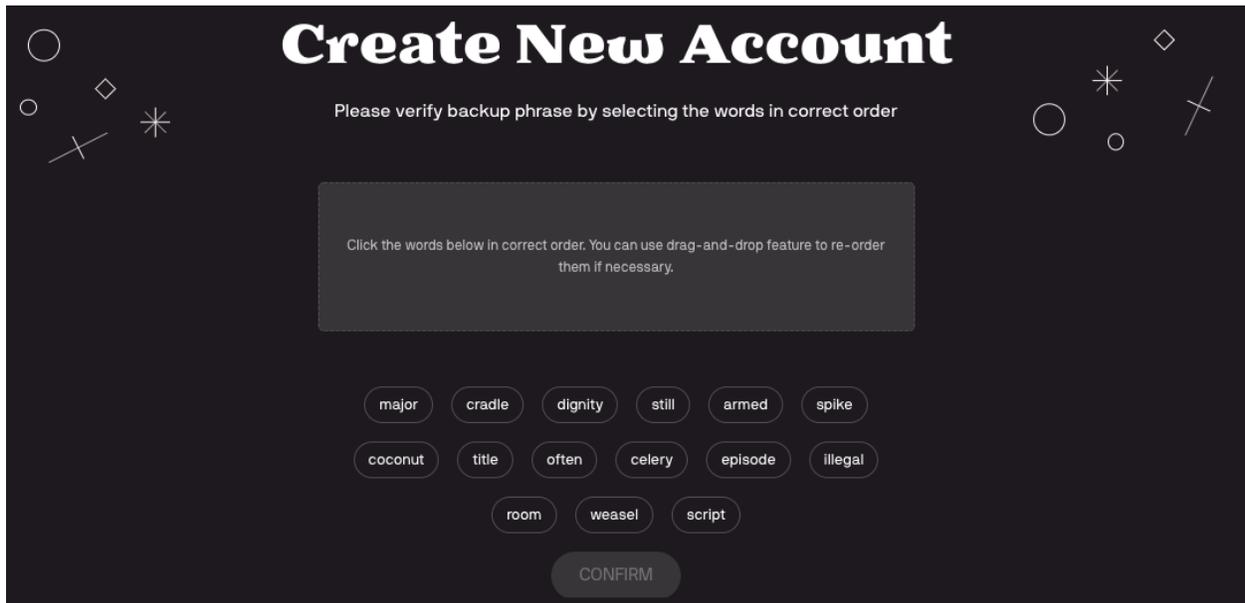
1. First, user will have to choose the account name along with a password for it. The name and password are not public, and will only be used for accessing the account later on.

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**).

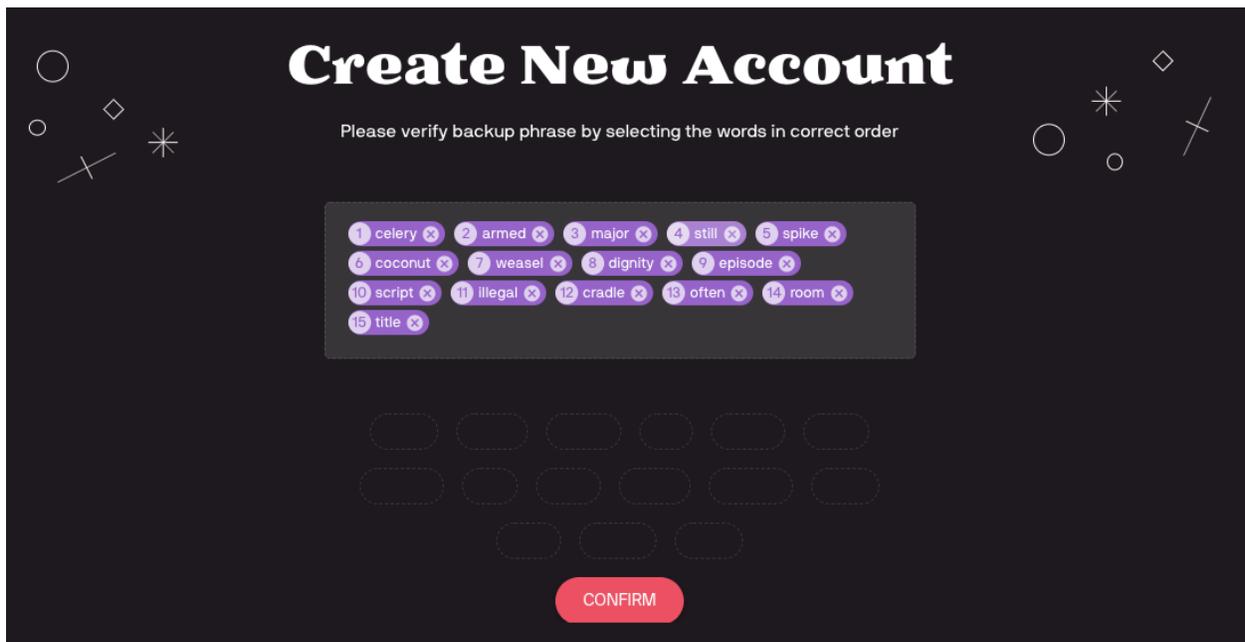
It is highly advised to print out or write down these words and store them in a safe place.

Important: Knowing these 15 words in correct order is the only way to retrieve the account if the password has been

lost.



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.

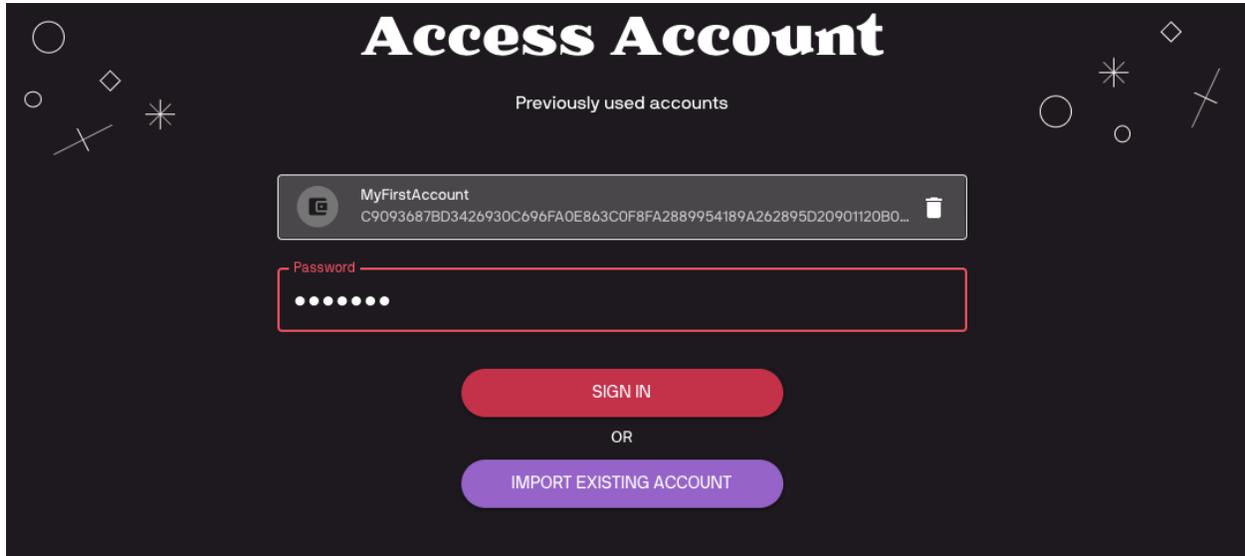


When all words are laid down in the correct order will the “Confirm” button be enabled.

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.

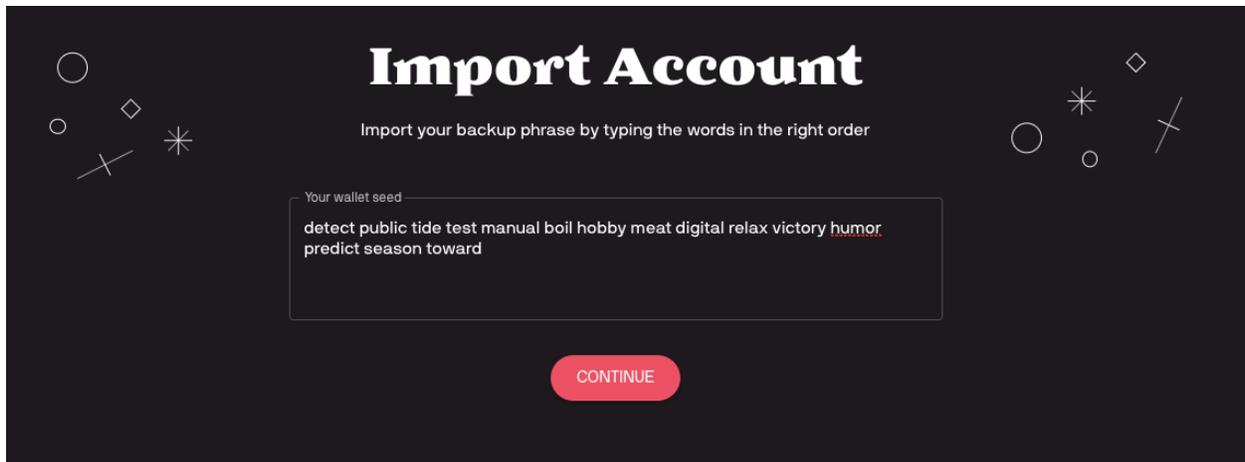


If the user wants to access an account on some other device (other than the one they have created the account on), they can use the “Import Existing Account”:

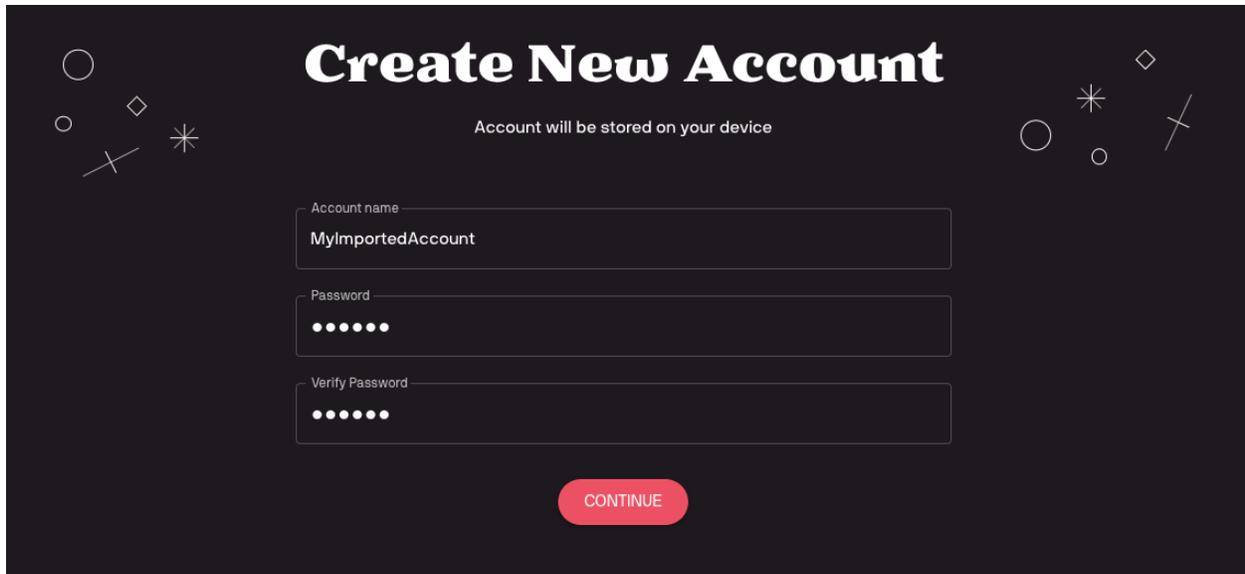
Importing existing account

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

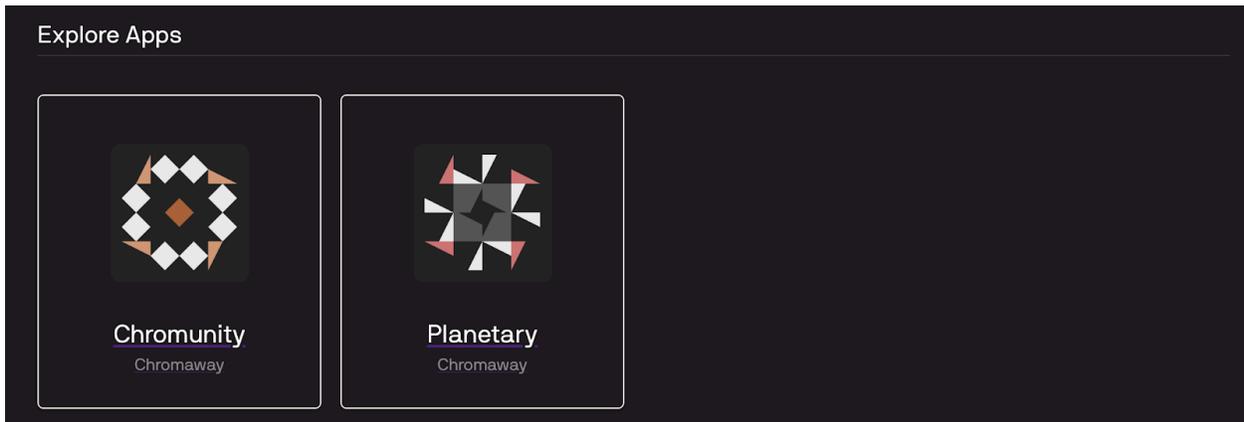
The Dashboard page is separated into 3 different sections: Chromia Accounts, Linked Apps and Explore Apps.

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 is basically it's 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-icons will always be unique as they are Account specific.

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)

Asset transfer logic is blockchain independent, which means that assets could be sent from any to any blockchain within the chromia ecosystem. In order to access the wallet section, one needs to select the account from the dashboard first.

The screenshot displays the Chromia Vault interface for the ChromaToken wallet. The interface is dark-themed and includes the following elements:

- Header:** Chromia VAULT logo and a back arrow.
- Wallet Info:** ChromaToken logo and the name 'Chromaway'.
- Assets Table:** A table showing the current asset balance.

Name	Amount
CHROMA	227
- Send Tokens Form:** A form with tabs for 'Send Tokens' and 'Receive Tokens'. It includes a 'Scan QR-Code' button, an 'Address' input field with a placeholder 'Paste the recipient address in the input above', an 'Application' dropdown, an 'Account' dropdown, an 'Amount' input field, an 'Asset' dropdown set to 'CHROMA', and a 'SEND' button.
- Transaction History Table:** A table showing recent transactions.

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	
- Footer:** 'Rows per page: 5' and '1-3 of 3' with navigation arrows.

Assets

The screenshot shows a dark-themed interface with the title 'Assets' at the top left. Below the title is a table with two columns: 'Name' and 'Amount'. A horizontal line is positioned below the column headers. The table contains one data row with 'CHROMA' in the 'Name' column and '227' in the 'Amount' column.

Name	Amount
CHROMA	227

Assets section is showing the list of all the available assets in that specific Account.

Sending assets

The screenshot displays the 'Send Tokens' interface. At the top, there are two tabs: 'Send Tokens' (active) and 'Receive Tokens'. A 'Scan QR-Code' button is located in the top right corner. The main area contains an 'Address' input field with the following text: 78967BAA4768CBCEF11C508326FFB13A956689FCB6DC3BA17F4B895CBB1577A3;EDA4B8221CD74411FA0891AC1297FD2C07E9AB37FBE1228845F635C8F6B91028. To the right of the address field, the 'Application' is 'ChromaToken' and the 'Account' is 'EDA4-1028'. Below the address field, there is a red 'SEND' button with a right-pointing arrow. At the bottom, there are two input fields: 'Amount' with the value '10' and 'Asset' with a dropdown menu showing 'CHROMA'.

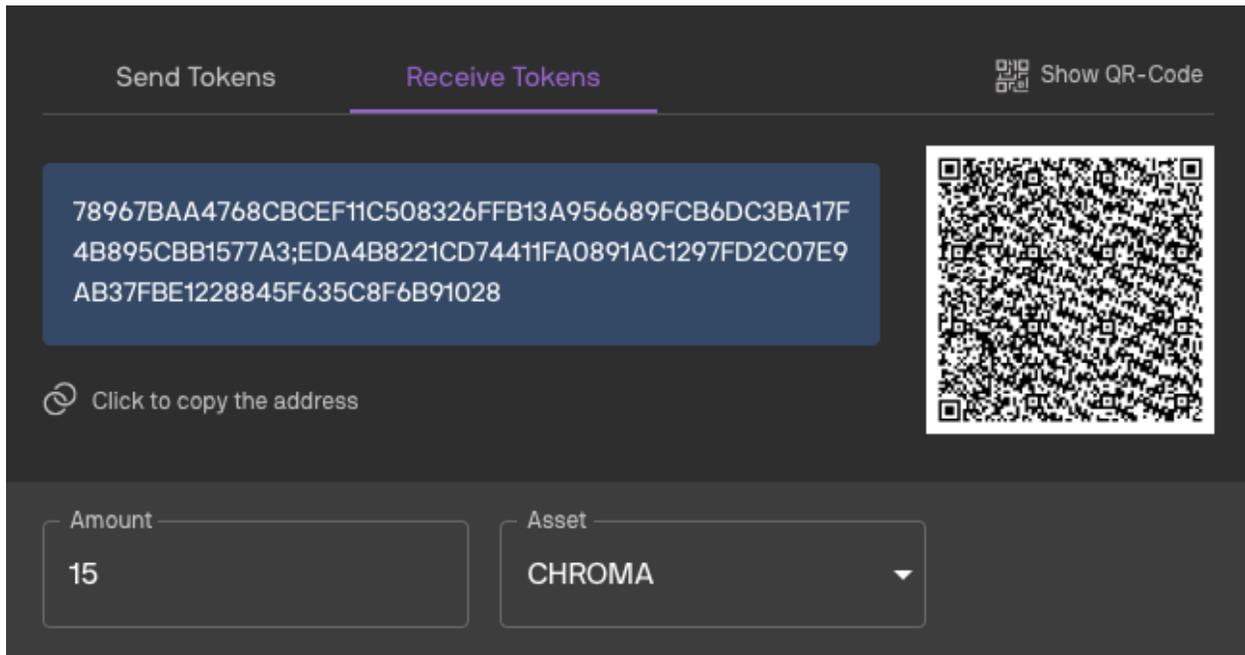
Assets are sent from the “Send Tokens” tab of the transfer section.

You can either enter the address manually (copy/paste) or use the scanner to scan 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



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

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	

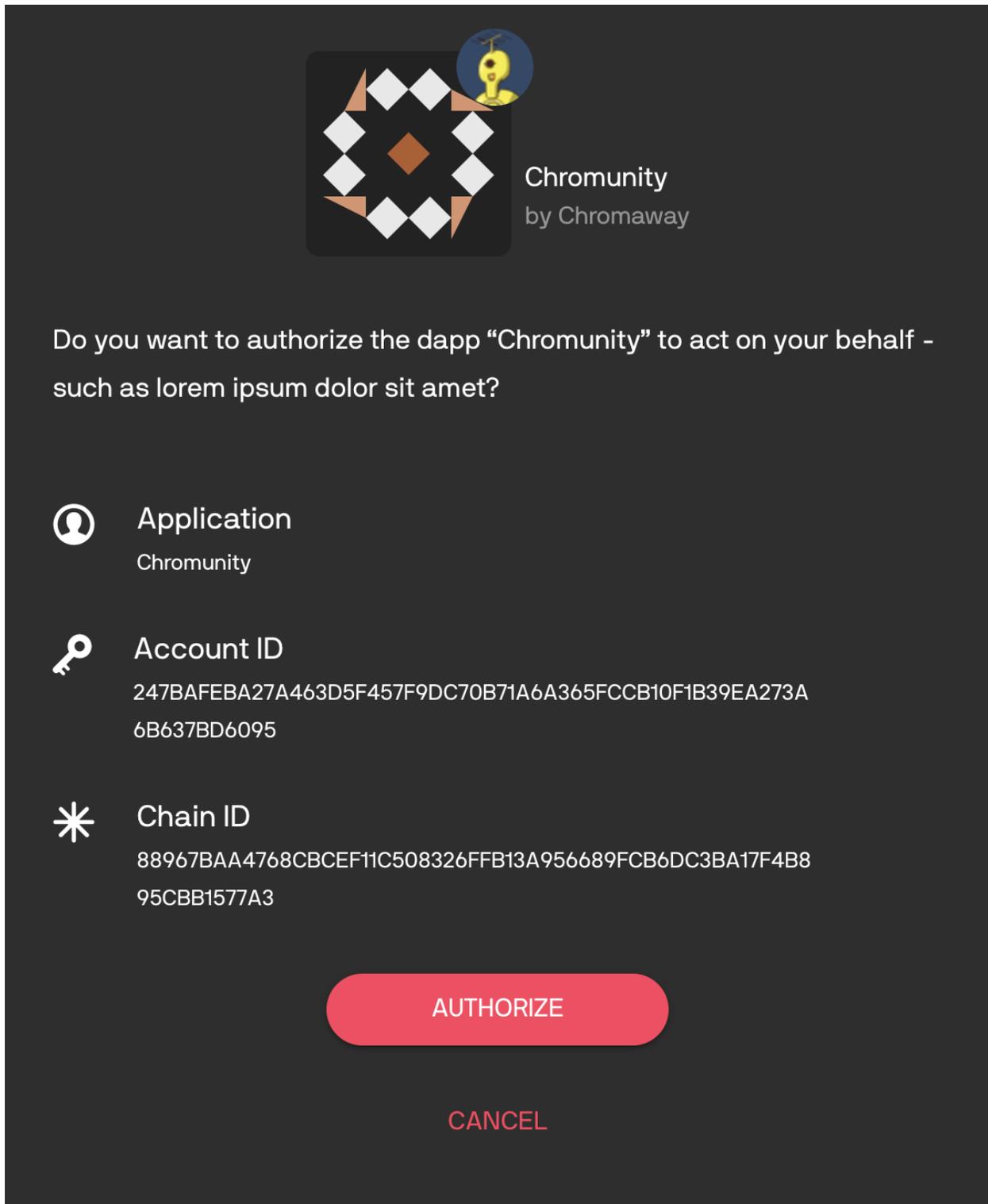
Rows per page: 5 | 1-3 of 3

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:



2.5.3 FT3 Module

FT3 is the recommended standards to handle tokens in Chromia ecosystem. It allow dapps to make full use of Chromia Vault, including Single Sign-On (SSO), assets transfer and visibility on the Vault’s dapp explorer.

FT3 consists of Rel module which contains blockchain logic, and client side lib which provides js API for interaction with the module.

Features

FT3 supports the following features:

- Account management
- Asset management and transaction history

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: **Account** and **Transfer**. The first flag specifies who can edit an account, and the later who can transfer the 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. Assets can be transferred inside and between different chains. Assets coming from outside of a chain are only accepted if source chains and assets are registered on the chain. The `asset` table contains list of registered assets (both native and the ones issued by other chains):

```
entity asset {
  id: byte_array;
  key id;
  key name;
  issuing_chain_rid: byte_array;
}
```

The balance table keeps track of an account's assets:

```
entity balance {
  key account, asset;
  mutable amount: integer = 0;
}
```

Project Setup

In this section, we explain how to setup a project to use FT3.

First lets clone the `ft3-lib` repository:

```
git clone https://bitbucket.org/chromawallet/ft3-lib.git
```

Checkout the development branch.

Create a new directory for your project, we will work inside this directory.

Blockchain side setup

Config project to use FT3

1. Inside the project create a `reII` directory with the following structure:

```
reII/
├── config/
└── lib/
```

2. From the cloned `ft3-lib`, copy `reII/ft3` directory into our `reII/lib` directory.
3. Inside `reII/config` create a `ft3_config.reII` file, and define a `ft3_config` object:

```
// reII/config/ft3_config.reII

object ft3_config {
  blockchain_name: text = <dapp_name>;
  blockchain_website: text = <dapp_website>;
  blockchain_description: text = <dapp_description>;
}
```

4. Create a `main.reII` file for the dapp inside `reII` directory, then include `ft3_config` and `ft3-lib`:

```
include "config/ft3_config";
include "lib/ft3/ft3_xc_basic_dev";
```

Warning: Make sure `ft3_config` is included before `ft3-lib`, because the lib has dependency on `ft3_config` object.

After this step `rell` directory should have the following structure:

```
rell/
├── config/
│   └── ft3_config.rell
├── lib/
│   └── ft3/
└── main.rell
```

Database setup

If you haven't already, follow the instruction here to setup a database: <https://rell.chromia.com/en/master/eclipse/eclipse.html#database-setup>

Node configuration

Next step is to add scripts, binaries and configuration which is needed for blockchain logic to run.

1. From `ft3-lib` copy `postchain` directory to our project's root directory.
2. Navigate to `postchain/config/nodes/`. Duplicate the `template` directory and rename it to `<dapp_name>` (or any name of your choice).
3. Update `postchain/config/nodes/my_dapp/blockchains/dapp/entry-file.txt` to specify path to `dapp main.rell` file (relative to `postchain` directory). In our case:

```
// entry-file.txt
../rell/main.rell
```

Now run:

```
postchain/bin/run-node.sh <dapp_name>
```

If everything is properly configured, you will soon see a success message printed to the console:

```
Postchain node launching is done
```

In subsequent runs, if we want to wipe old database, we can add the `-W` option:

```
postchain/bin/run-node.sh <dapp_name> -W
```

With that the blockchain side is ready, we can go on to the client side.

Client side setup

Create a client directory in project root and run `npm init` inside it (or bootstrap a project using a generator, e.g. `create-react-app`).

Add dependencies to the node project:

```
npm i --save ft3-lib
npm i --save postchain-client
```

Add other libraries to your liking.

Directory Chain

In Chromia ecosystem, the **directory chain** is responsible for keeping track of all chains. FT3 will connect to directory chain to get connection info of a chain to which the lib wants to connect.

The FT3 client lib contains a *DirectoryService* interface for this interaction.

Since directory chain is not implemented yet, we have to implement a *DirectoryService* and provide a list of chains for FT3 to access.

The simplest way to define directory chain is to extends *DirectoryServiceBase* class, which should be enough in most of the cases. Inside client directory create a `/lib/directory-service.js` file:

```
// client/lib/directory-service.js
import { DirectoryServiceBase, ChainConnectionInfo } from 'ft3-lib';

const chainList = [

  // Our local chain
  new ChainConnectionInfo(
    Buffer.from(
      '0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF',
      'hex'
    ),
    'http://localhost:7740'
  ),
  /*
  We can add other chains to the list in the future:
  new ChainConnectionInfo(
    Buffer.from('<chain1RID>', 'hex'),
    'http://chain1URL'
  ),
  new ChainConnectionInfo(
    Buffer.from('<chain2RID>', 'hex'),
    'http://chain2URL'
  ),
  new ChainConnectionInfo(
    Buffer.from('<chain3RID>', 'hex'),
    'http://chain3URL'
  )
  */
];

export default class DirectoryService extends DirectoryServiceBase {
  constructor() {
```

(continues on next page)

(continued from previous page)

```

    super(chainList);
  }
}

```

Note: Instead of defining the `chainList` in the class file, it should be done in a configuration file of the app.

Conclusion

That concluded the project setup process. After setup, our project should look like this:

```

root
├── client/
│   ├── lib/
│   │   └── directory-service.js
│   └── ...
├── poschain/
│   ├── config/
│   │   └── nodes/
│   │       └── my_dapp/
│   └── ...
└── rell/
    ├── config/
    │   └── ft3_config.rell
    ├── lib/
    │   └── ft3/
    └── main.rell

```

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 Blockchain object which is used to interact with the blockchain:

```

// /client/index.js
import { Blockchain } from 'ft3-lib';
import DirectoryService from './lib/directory-service';

const blockchainRID =
↪ '0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF';
const chainId = Buffer.from(
  blockchainRID,
  'hex'
);

```

(continues on next page)

(continued from previous page)

```
const blockchain = await Blockchain.initialize(
  chainId,
  new DirectoryService()
);
```

Note: Chain id is specified in the config file `postchain/config/nodes/<dapp_name>/blockchains/ft3/brid.txt`. Here we use the default value `'0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF'`.

Details of the initialized chain can be accessed by accessing `info` (instance of `BlockchainInfo`) 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 are set in the `ft3_config.rell` 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 user has this type of authDescriptor
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(authDescriptorId);
```

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 case 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`.

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.

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.

Adding authentication descriptor

```
const newAuthDescriptor = new SingleSignatureAuthDescriptor(
  pubKey,
  [FlagsType.Account, FlagsType.Transfer]
);
const account = await session.getAccountById(accountId);
await account.addAuthDescriptor(newAuthDescriptor);
```

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()
```

(continues on next page)

(continued from previous page)

```
.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

```
record player {
    first_name: text;
    last_name: text;
}

operation some_op(player) {
    ...
}
```

RelI Integration

Defining DApp user class

The `ft3.account` class provided by `ft3` module can be used for account management.

An example dapp's user account could be defined as:

```
// user.rell
entity user {
  first_name: text;
  last_name: text;
  age: integer;
  key account: ft3.account;
}
```

In this user model, there is no public key or user ID. Those details are provided by `ft3.account`.

`ft3.account` has an `id` property which uniquely identifies an account, and access is controlled by `ft3.account_auth_descriptor` which include user public key.

The underlying structure of `ft3.account` and `ft3.account_auth_descriptor` is explained in *FT3 Features*.

Note: It should be noted that this `user` entity is a dapp-defined user, and has no relationship with `ft3-lib`'s `User` class mentioned in earlier section.

Now let's add user entity to the project. First we create `user.rell` in `rell` directory and add user entity defined above, and then include user file in `main.rell`:

```
// main.rell
include "config/ft3_config";
include "lib/ft3/ft3_xc_basic_dev";
include "user";
```

After user class is defined, the next step is to define operation used to create an instance of user:

```
operation create_user(
  first_name: text,
  last_name: text,
  age: integer,
  user_auth: ft3.auth_descriptor
) {
  val account_id = ft3.create_account_with_auth(user_auth);
  create user (
    first_name,
    last_name,
    age,
    ft3.account @ { account_id }
  );
}
```

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 user table:

```
postchain/bin/run-node.sh <dapp_name> -W
```

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 {
  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(
  chainId,
  new DirectoryService()
);
const session = blockchain.newSession(user);

await session.call(op(
  'create_user',
  'John',
  'Doe',
  30,
  user.authDescriptor
));
```

We can check if create_user operation is executed successfully by connecting to postchain database and executing:

```
select * from dapp_name."c0.user";
```

Single Sign-on (SSO)

SSO allows a user to login to different applications with a single account. In order for SSO to work, application has to be integrated with a SSO service.

In Chromia ecosystem the role of a SSO service is handled by Chromia Vault, though it can be easily implemented in any other FT3 application thanks to FT3 module's flexible authentication model.

As already said access to a FT3 account is controlled with authentication descriptors. So if an auth descriptor with a Vault account's public key is added to dapp's account, Vault will have control over the dapp account and it will be able to edit authentication descriptors.

When user wants to SSO into a dapp, dapp just has to generate a new key pair, and ask Vault to add new auth descriptor with that keypair to the dapp's account. The user (or rather, the user's current device) can now use that new key pair to perform transactions for the account. Such keypairs are disposable and can safely be discarded or replaced at user's discretion.

Register flow

A standard register flow is as follow:

Step 1. At *Dapp*:

1. User click on 'Register' button.
2. Redirect to vault to collect vault account's pubkey to be added to ft3 account during registration.
 - An auth descriptor with Vault's pubkey is added for SSO (add new auth descriptors).
 - Another auth descriptor (created by dapp) is used to interact with the dapp (sign transactions).
 - Query parameter:
returnUrl - dapp url which will be opened after collecting vault account pub key

When using Vault as SSO service, the redirect url should looks like this: `https://wallet-v2.chromia.dev/?route=/link-account&returnUrl=<dapp_address>`.

Step 2. At *Vault*:

1. User login to a vault account.
2. Vault get user's pubkey.
3. Redirect back to dapp (using returnUrl) with query paramter:
pubkey - vault account's public key which will be added to auth descriptor.

Step 3. At *Dapp*:

1. Generate new key pair for user auth descriptor.
2. Create auth descriptor with generated pub key (user auth descriptor).
3. Read vault account pub key from query parameter.
4. Create auth descriptor with vault account pub key (vault auth descriptor).
5. Register dapp user account with user auth descriptor, and add vault auth descriptor for the newly created account.

And now js code sample. First we open the Vault to collect Vault account's pub key:

```
const returnUrl = encodeURIComponent('http://my-dapp-account-creation-url');
window.location.href = `https://wallet-v2.chromia.dev/?route=/link-account&returnUrl=${returnUrl}`;
```

After user logs into their Vault account, Vault then redirects them to returnUrl page in the dapp where following logic has to be executed:

```
import { util } from 'postchain-client';
import {
  op,
  Blockchain,
  SingleSignatureAuthDescriptor,
  FlagsType,
  User
} from 'ft3-lib';

// The parse function parses query parameter into an object
const { pubkey } = parse(location.search || {});
```

(continues on next page)

(continued from previous page)

```

const vaultAuthDescriptor = new SingleSignatureAuthDescriptor(
  pubKey,
  [FlagsType.Account, FlagsType.Transfer]
);

const keyPair = util.makeKeyPair();
const user = new User(
  keyPair,
  new SingleSignatureAuthDescriptor(
    keyPair.pubKey,
    [FlagsType.Account, FlagsType.Transfer]
  )
);

const blockchain = await Blockchain.initialize(
  chainId,
  new DirectoryService()
);

await blockchain.call(
  op(
    'create_user',
    firstName,
    lastName,
    age,
    user.authDescriptor,
    vaultAuthDescriptor
  ),
  user
);

```

On the blockchain, we will have to update `create_user` operation to handle 2 auth descriptors:

```

operation create_user(
  first_name: text,
  last_name: text,
  age: integer,
  user_auth: ft3.auth_descriptor,
  vault_auth: ft3.auth_descriptor
) {
  val account_id = ft3.create_account_with_auth(user_auth);
  create user (
    first_name,
    last_name,
    age,
    ft3.account @ { account_id }
  );

  val account = account @ { .id == account_id };
  ft3._add_auth_descriptor(account, vault_auth);
}

```

Note: Even though it make more sense for the account to be created using `vault_auth` (because `user_auth` is a disposable keypair), it should be noted that while creating an account, `ft3` create `account_id` by hashing the auth descriptor used for creation.

Because `account_id` is a key attributes of `account`, if you use `vault_auth` to create the account and an user try to register a second account, the same `vault_auth` as the first will be used and thus fails the transaction.

Login flow

Step 1. At *Dapp*:

1. User click on 'Login with wallet' button to login to a dapp account.
2. Generate a new key pair and save key pair to local storage.
3. Redirect to user the vault to authorize generated key pair. Query parameters:
 - `dappId` - ID of the dapp (blockchain RID)
 - `accountId` - dapp account ID
 - `pubkey` - generated public key
 - `successAction` - vault redirects to this URL after successful authorization

Step 2. At *Vault*:

1. Userlogin to vault.
2. Vault display authorization form with dapp account details.
3. User click authorize.
4. Vault connects to the dapp chain and adds authorization descriptor.
5. Vault redirect user to the URL provided in `successAction` query parameter.

Step 3. At *Dapp*:

1. Read stored key pair from Step 1.
2. Complete dapp login flow and start controlling account with the keypair.

And again here's a basic code which does the steps needed on a dapp side:

```
import { util } from 'postchain-client';
import DirectoryService from './lib/directory-service';

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

localStorage.setItem('keyPair', JSON.stringify({
  pubKey: pubKey.toString('hex'),
  privKey: privKey.toString('hex')
}));

const returnUrl = encodeURIComponent('http://my-dapp-login-success-url');

const href = `https://wallet-v2.chromia.dev/?route=/authorize&dappId=${chainIdString}&
↪accountId=${accountId}&pubkey=${pubKey}&successAction=${returnUrl}`;

window.location.href = href;
```

And then on the page where we are redirected after successful authorization (step 3) we verify auth descriptor is added to our ft3 account:

```

const { pubKey, privKey } = JSON.parse(localStorage.getItem('keyPair'));

const keyPair = {
  pubKey: Buffer.from(pubKey, 'hex'),
  privKey: Buffer.from(privKey, 'hex')
};

const authDescriptor = new SingleSignatureAuthDescriptor(
  keyPair.pubKey,
  [FlagsType.Account, FlagsType.Transfer]
);

const user = new User(keyPair, authDescriptor);
const blockchain = await Blockchain.initialize(
  Buffer.from(chainIdString, 'hex'),
  new DirectoryService()
);

const accounts = await blockchain.getAccountsByParticipantId(
  keyPair.pubKey,
  user
);

const isAdded = accounts.some(({ id_ }) => (
  id_.toString('hex').toUpperCase() === accountId.toUpperCase()
));

if (isAdded) {
  // do dapp specific login, i.e. initialize dapp user object
} else {
  // display login error
}

```

Congratulations! Your dapp is now fully FT3-integrated!

2.6 Upgrading to Rel 0.10

There are two kinds of breaking changes in Rel 0.10.0:

1. Rel 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.

2.6.1 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 uses the root module as the main module, so make sure you have it and import all required modules there.

2.6.2 Details

migrate-v0.10.sh tool

The tool can be found in the `postchain-node` directory of a Rel 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 `.rell` 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>
            </dict>
          </entry>
          <entry key="app.bar">
```

(continues on next page)

(continued from previous page)

```

        <dict>
          <entry key="x">
            <int>123456</int>
          </entry>
          <entry key="y">
            <string>Hello!</string>
          </entry>
        </dict>
      </entry>
    </dict>
  <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 ReII 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>

```

(continues on next page)

(continued from previous page)

```
</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

2.7 Eclipse IDE

2.7.1 Installation

Prerequisites

- Java 8+.
- To be able to run a Postchain node: PostgreSQL 10.

Using a full Rell Eclipse IDE bundle

Rell Eclipse IDE can be downloaded here: <https://www.chromia.dev/rell-eclipse/index.html>.

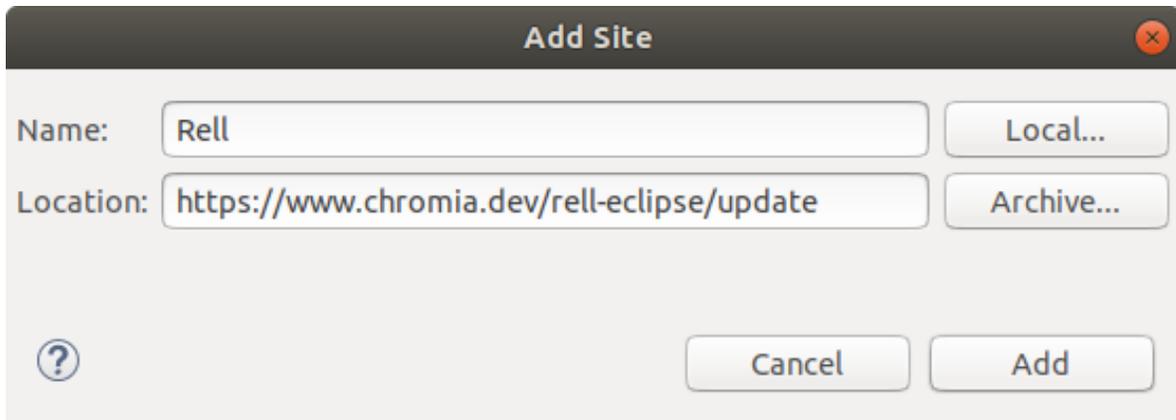
Download an archive for your OS.

- Linux: unpack, run `eclipse`
- Windows: unpack, run `eclipse.exe`
- MacOS: open the DMG image, run or install Eclipse.

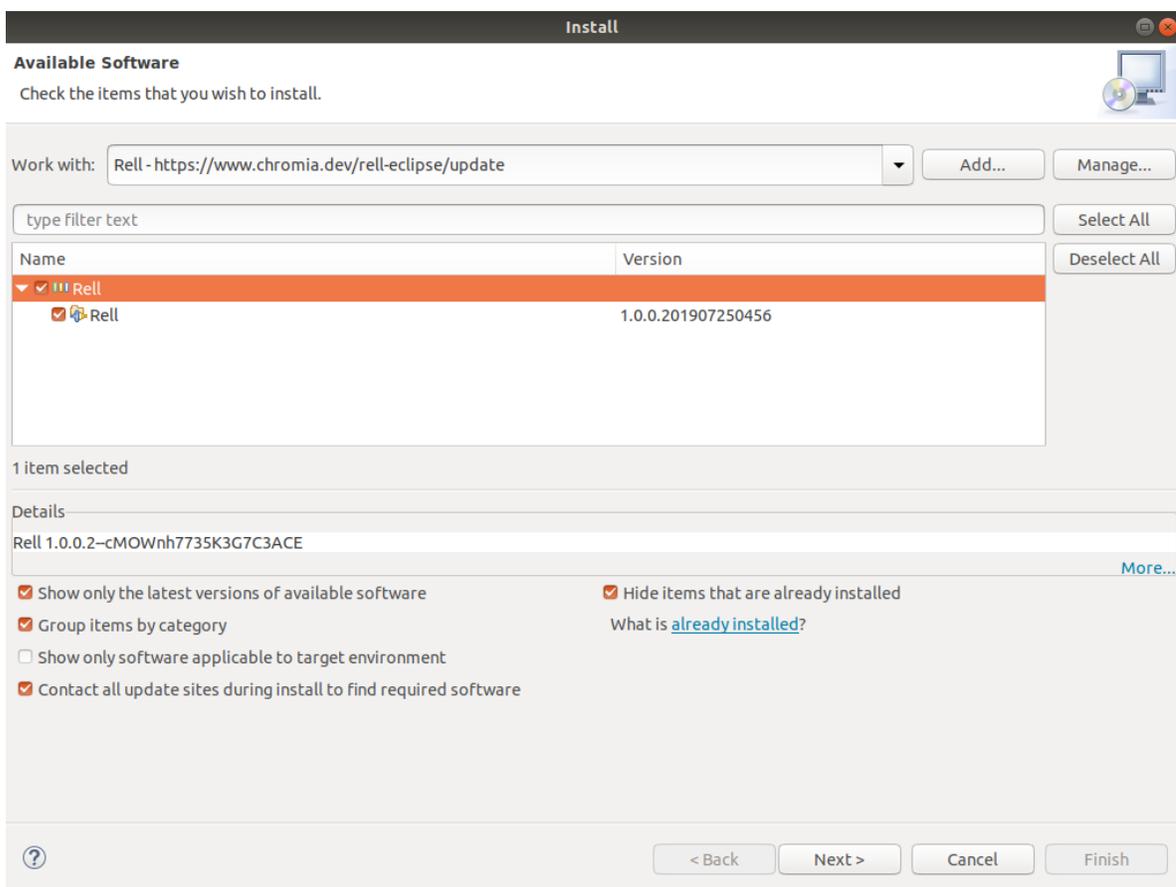
Adding Rell plugin to an existing Eclipse IDE

If you already have an Eclipse IDE (for example, Eclipse IDE for Java), the Rell plugin can be added to it.

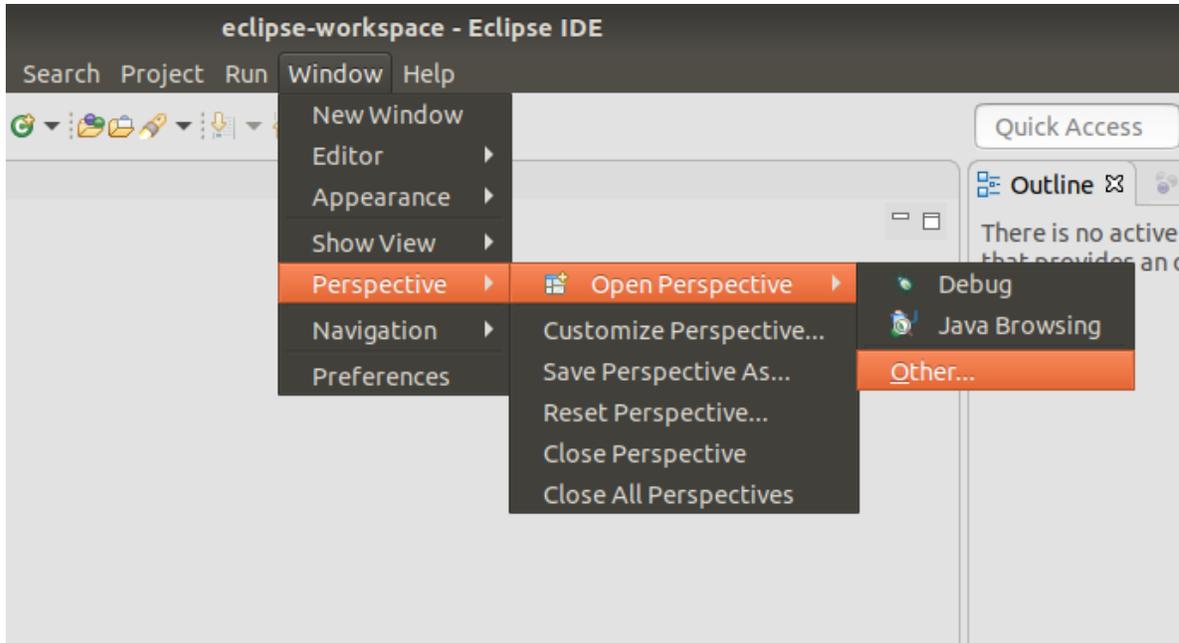
1. Go to the menu **Help - Install New Software...**
2. In the Install dialog, click **Add...**, then type:
 - Name: Rell
 - Location: <https://www.chromia.dev/rell-eclipse/update>



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



4. When seeing a warning “You are installing software that contains unsigned content”, click **Install anyway**.
5. Click **Restart Now** when asked to restart Eclipse IDE.
6. Switch to the RelI perspective. Menu **Window - Perspective - Open Perspective - Other...**, choose **RelI**.



Next step is to create a RelI project (see *Hello World Program*).

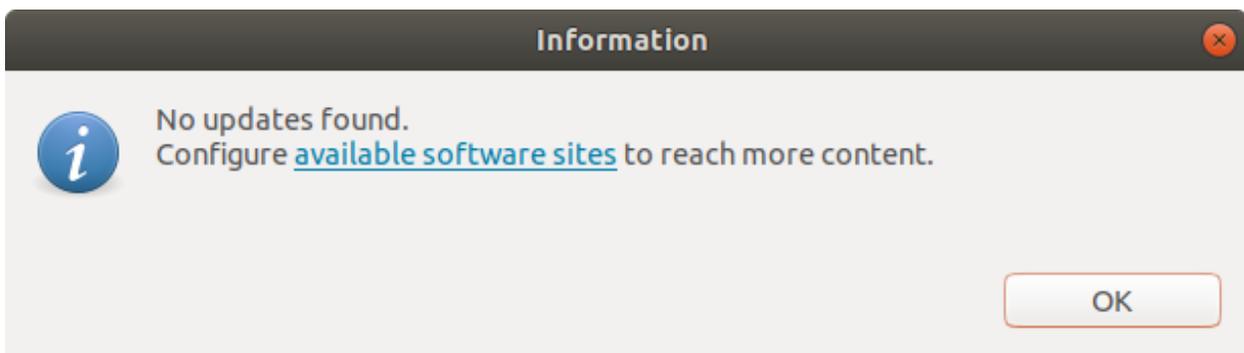
How to update the RelI plugin

When a new version of the RelI plugin is released, it has to be updated in Eclipse. If RelI 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.

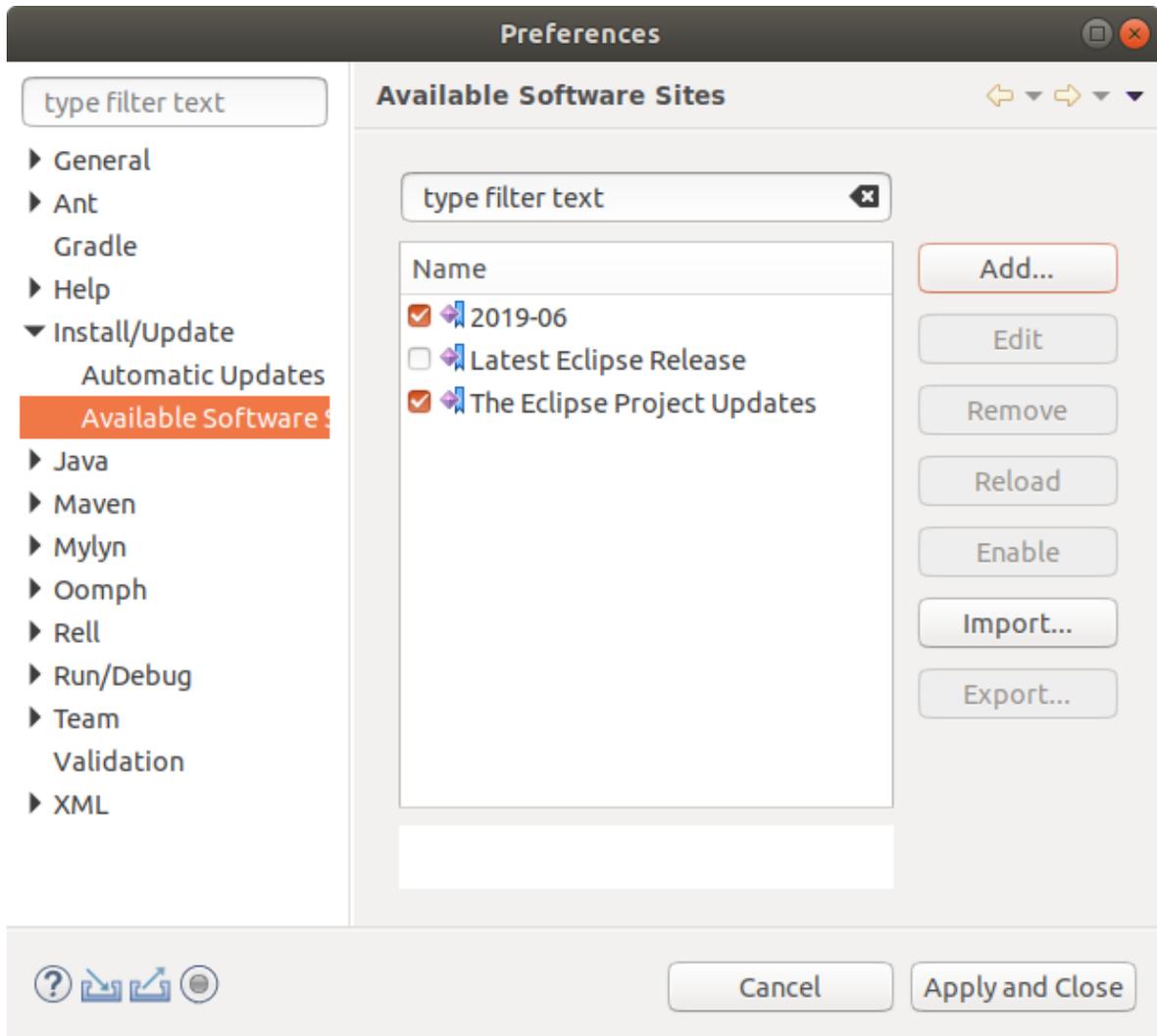
To manually check for updates:

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

If “No updates found” message is shown, check that RelI 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: Rell
- Location: <https://www.chromia.dev/rell-eclipse/update>

and click **Add**.

If Rell 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 Rell plugin must be already up-to-date.

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 Rel 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:

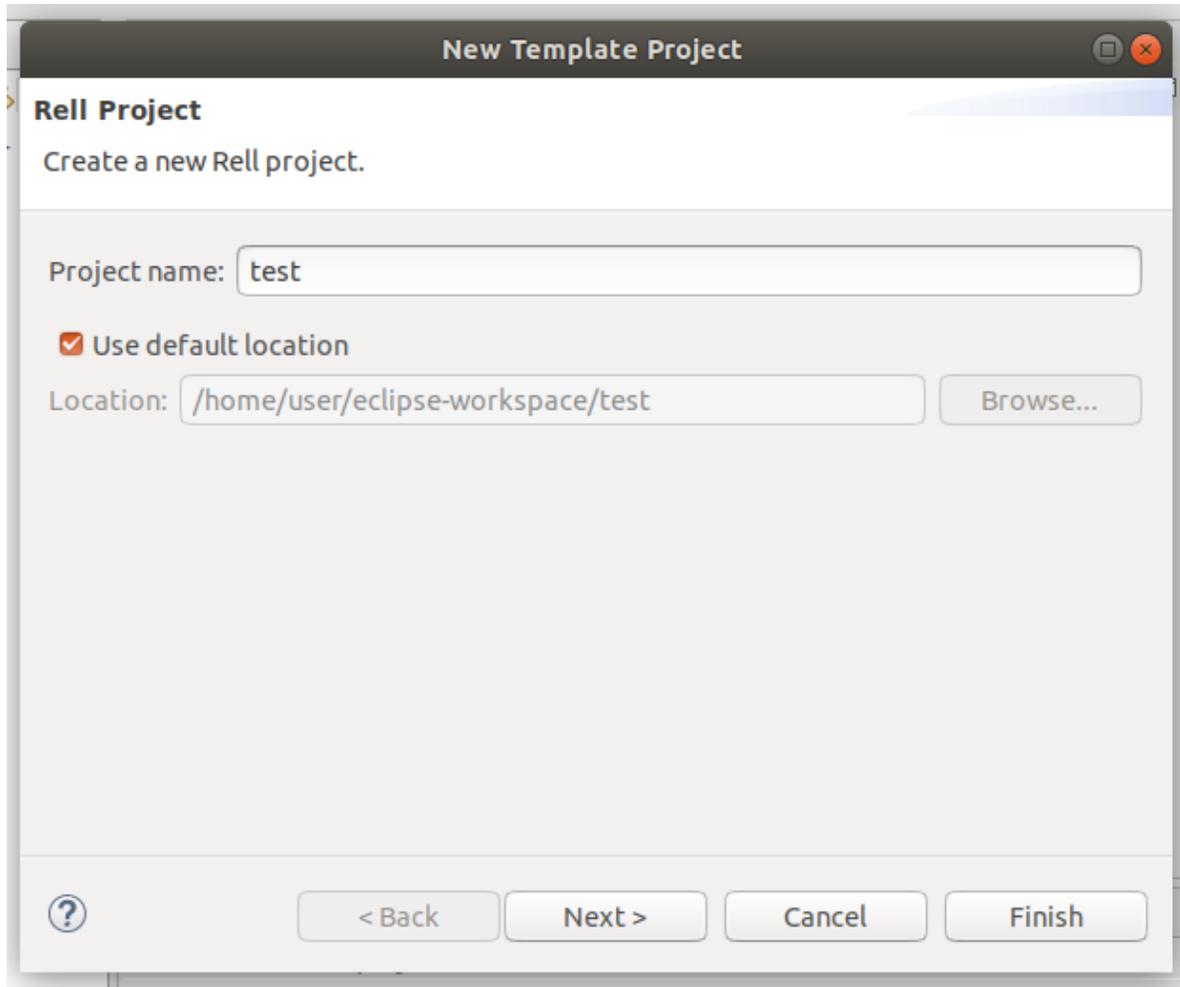
```
brew install postgresql  
brew services start postgresql  
createuser -s postgres
```

Prepare a Rel database:

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

2.7.2 Hello World Program

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**.



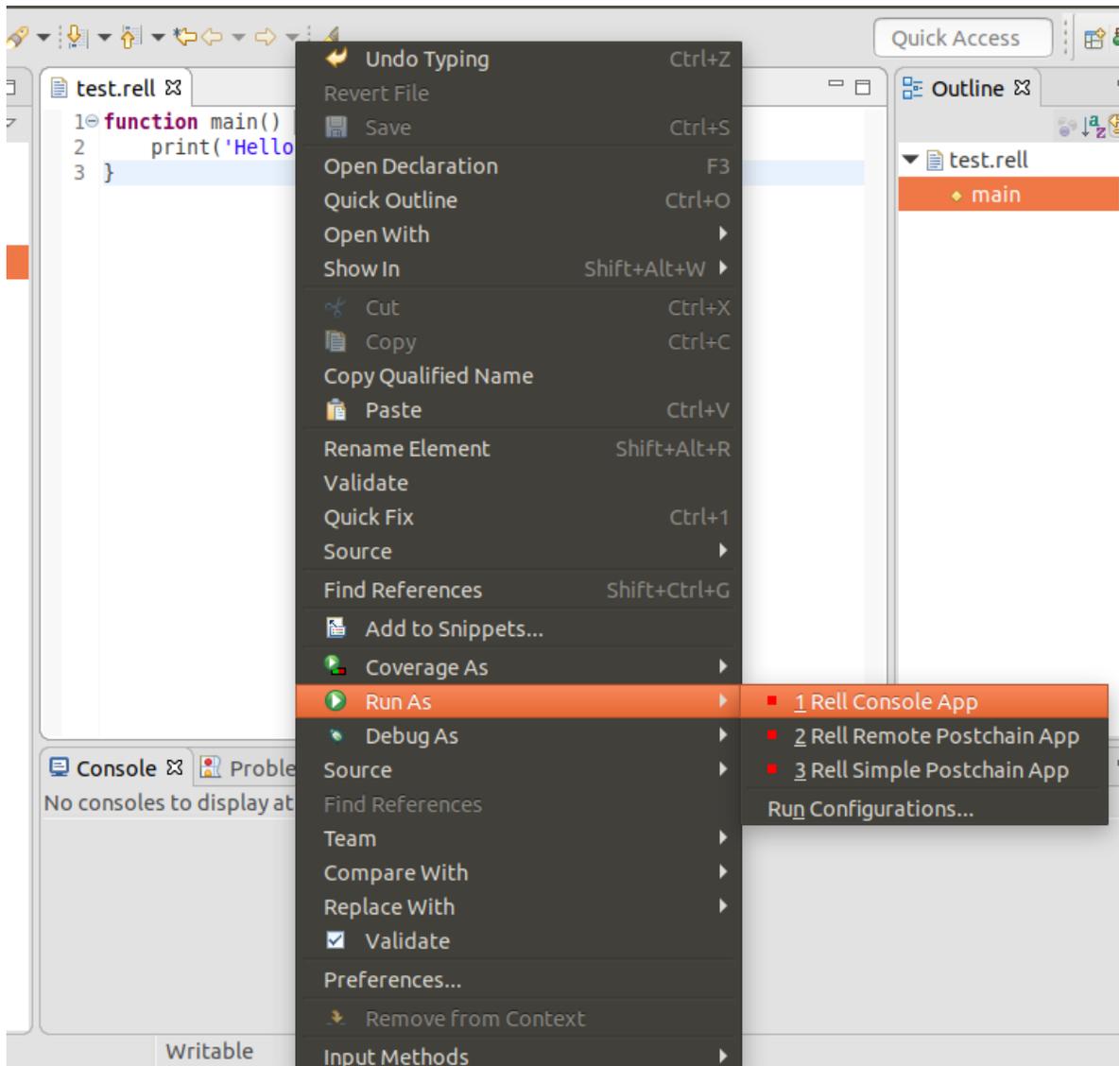
A Rell project with a default directory structure shall be created.

3. Create a Rell file:
 - Right-click on the `src` folder and choose **New - File**.
 - Enter a file name `test.rell` and click **Finish**.
4. Write the code in the editor:

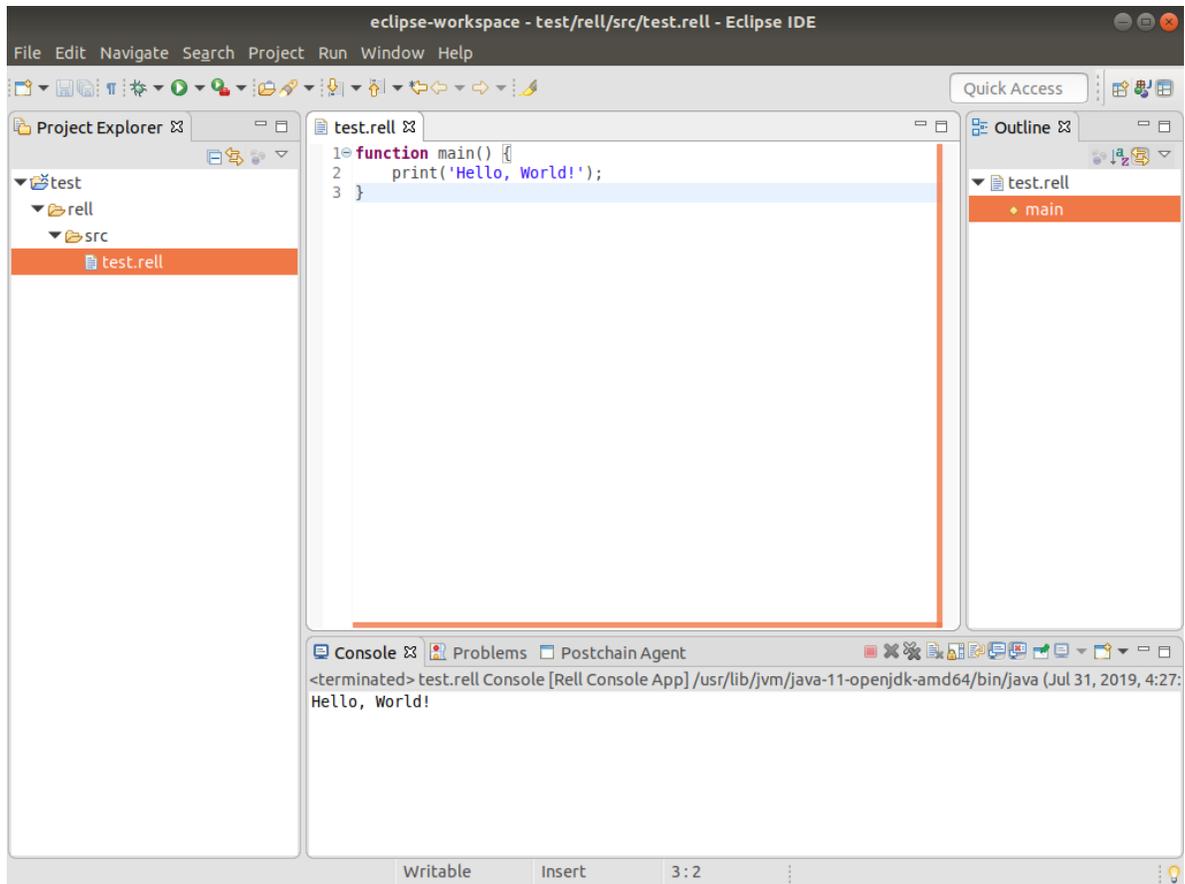
```
function main() {  
    print('Hello, World!');  
}
```

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

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



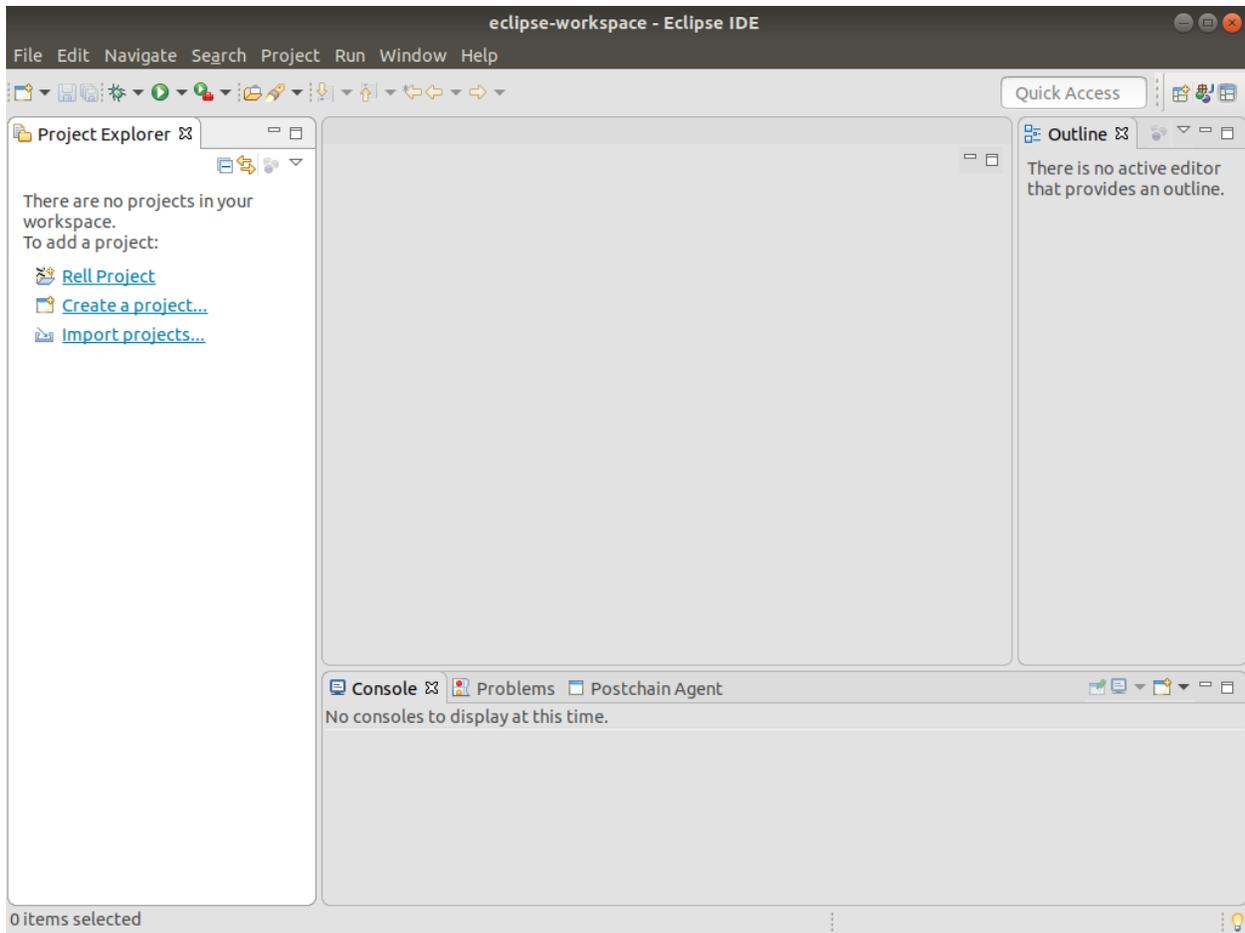
6. The output “Hello, World!” must be shown in the Console view.



2.7.3 IDE Overview

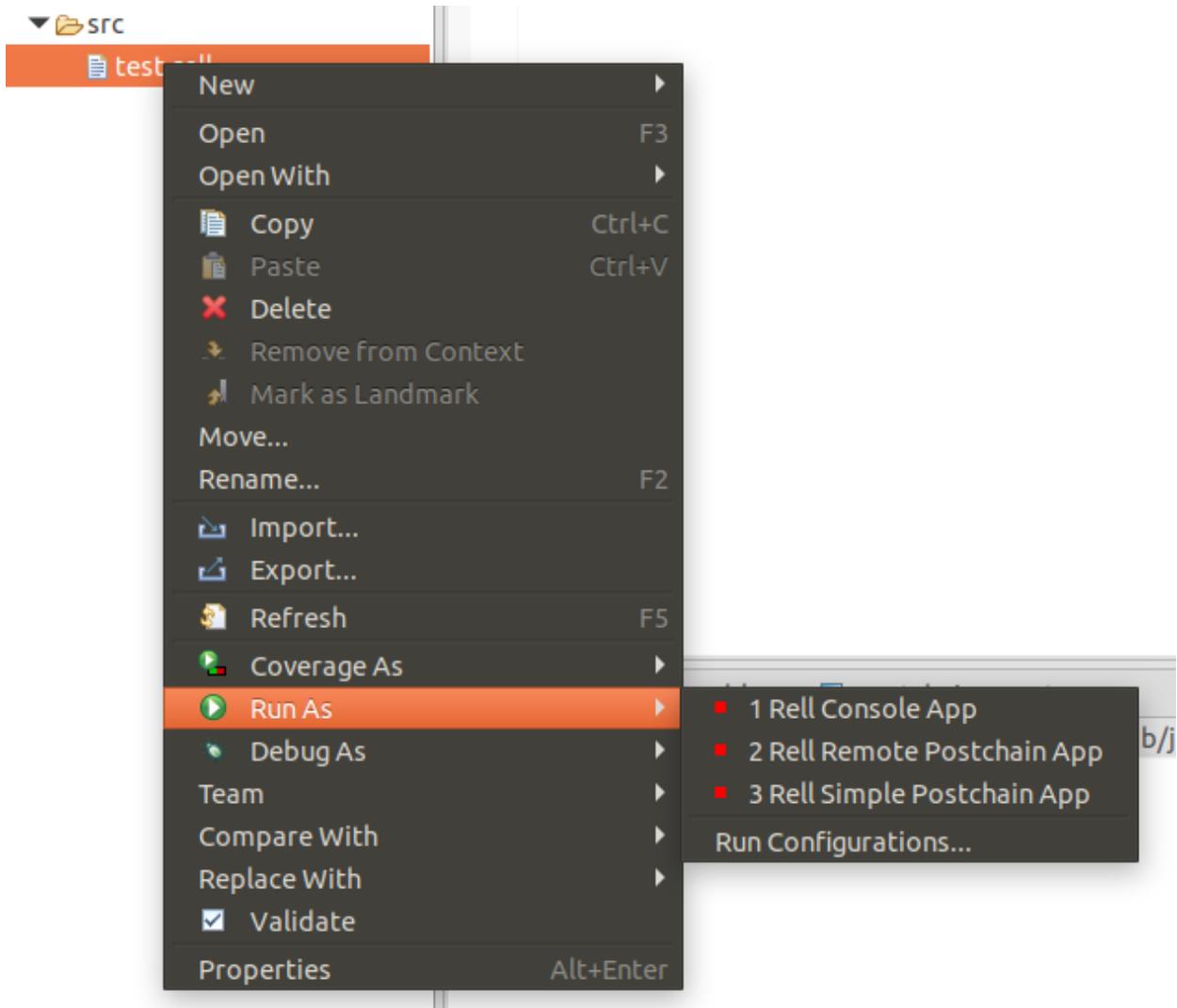
Eclipse IDE window consists of different views. Every view has its own tab. By default, Rel IDE has following views:

- **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 Rel file.

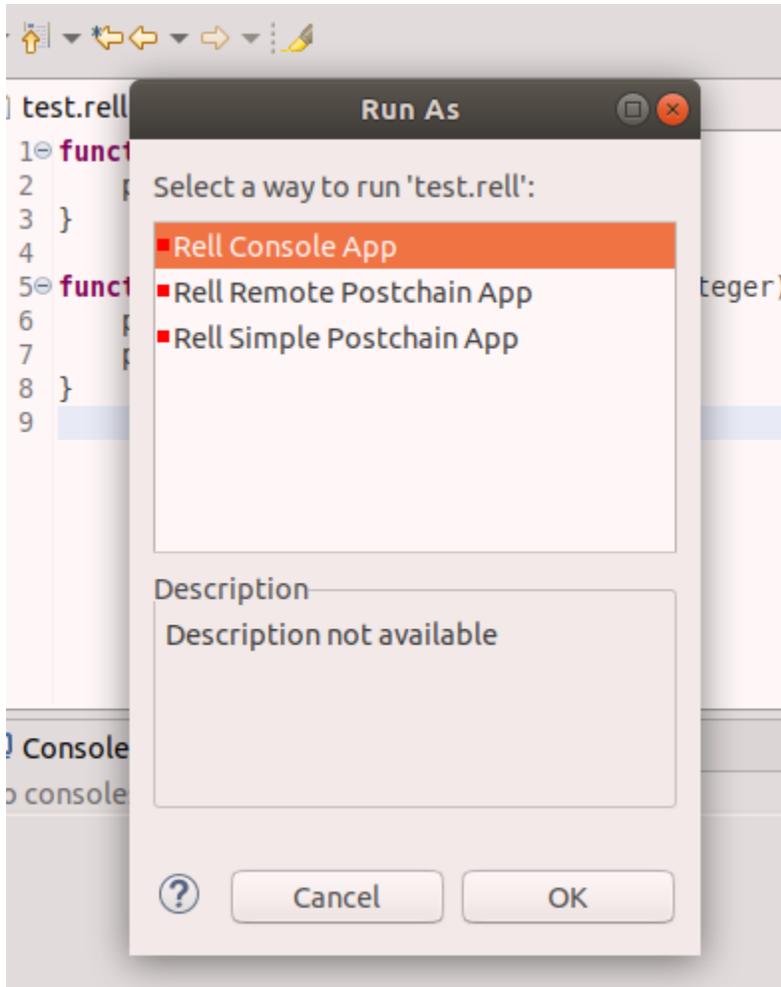


2.7.4 Running Applications

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).



ReII 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.

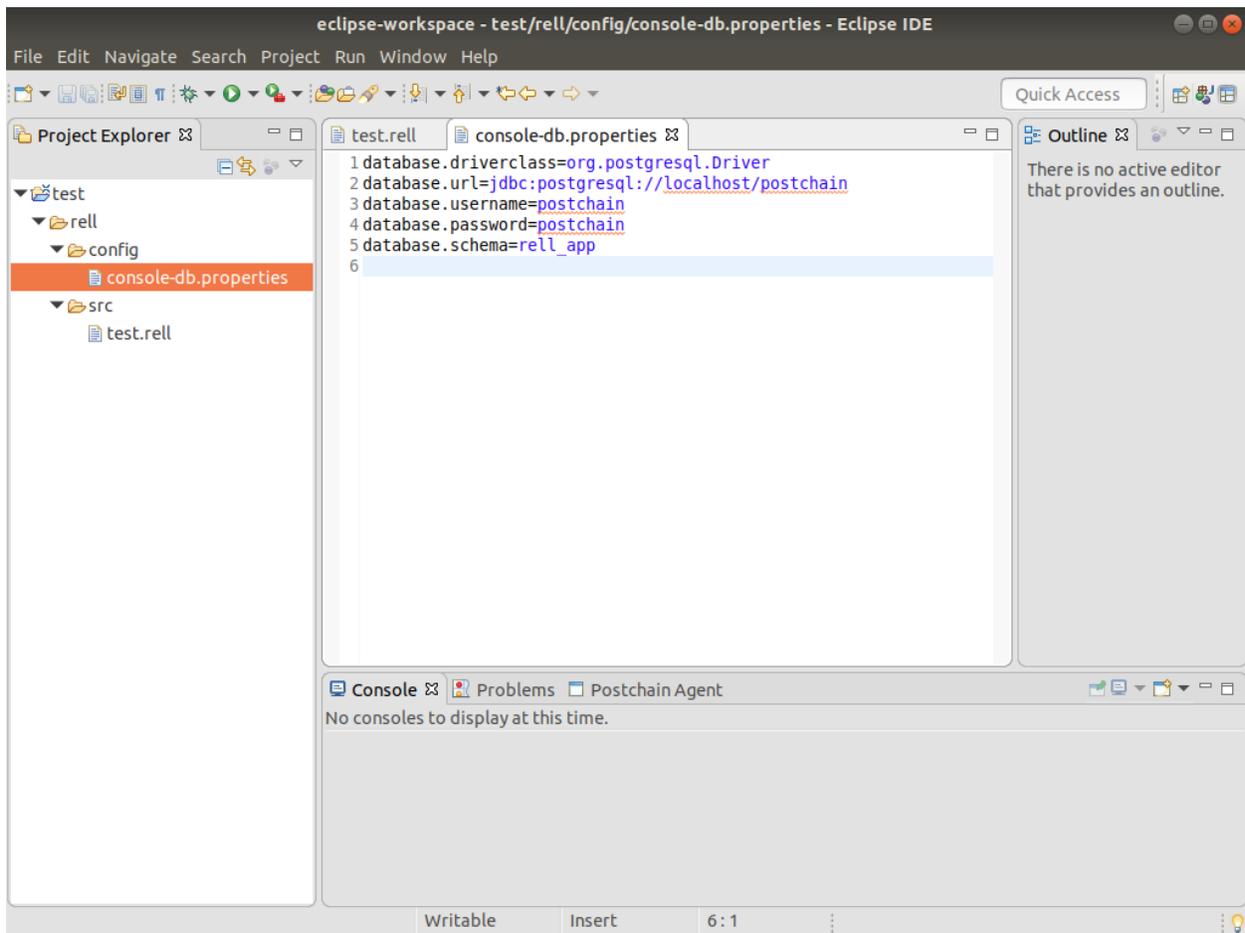
To run a console app with a database connection, there must be a file called `console-db.properties`, `db.properties` or `node-config.properties` in the directory of the ReII file or in the `rell/config` directory of the project. The file shall contain database connection settings. For example:

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

(continues on next page)

(continued from previous page)

```
database.password=postchain
database.schema=rell_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.

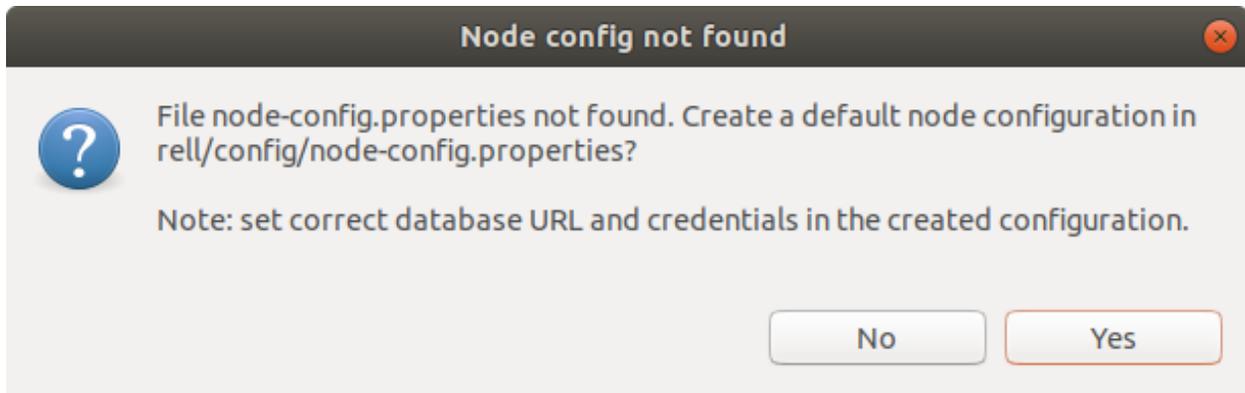
How to prepare a database is described in the [Database Setup](#) section.

Rel Simple Postchain App

Starts a Postchain node running the given Rel module.

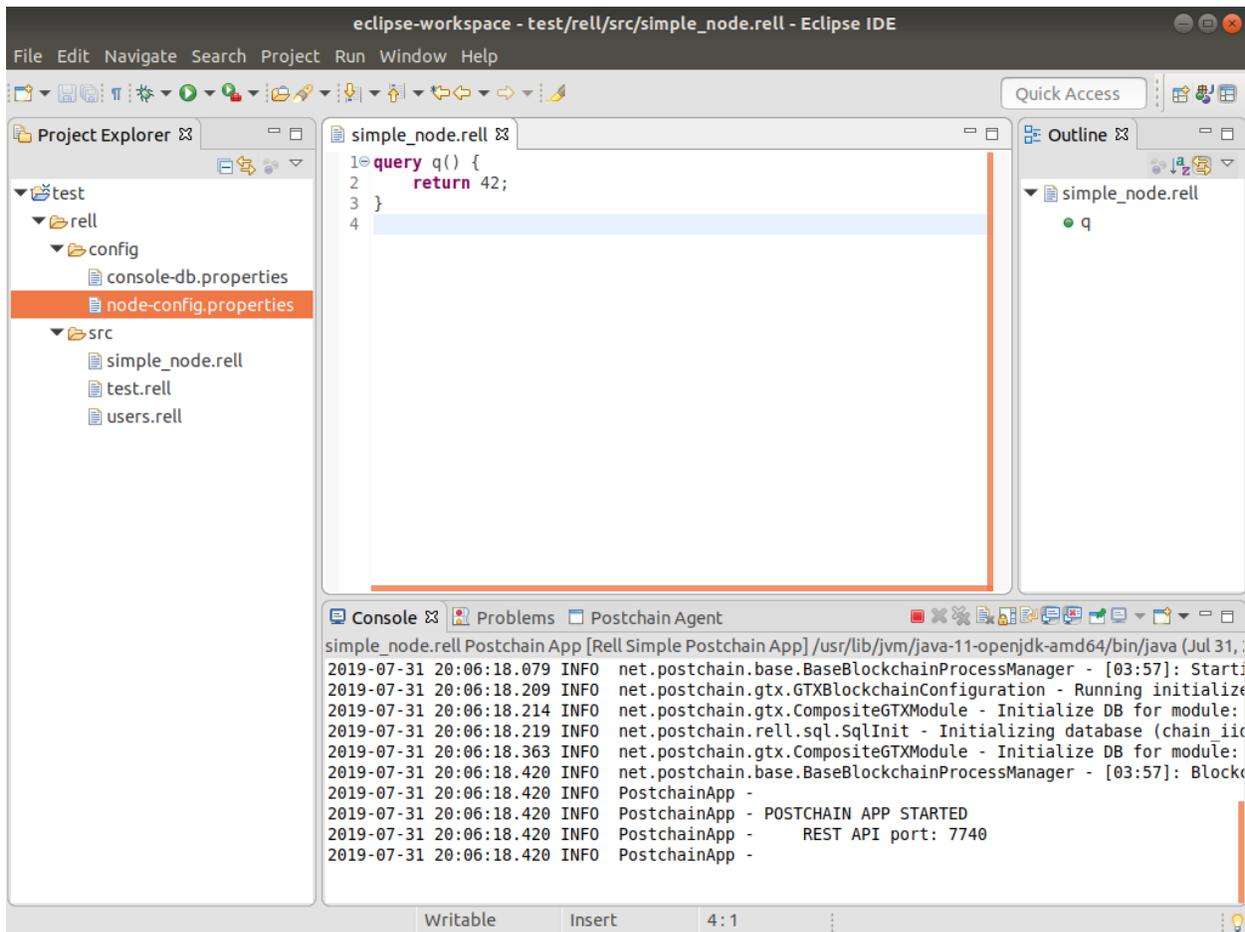
A database connection and other standard Postchain node properties must be specified in a `node-config.properties` file. The file is searched in the directory of the selected Rel file and then in the `rell/config` directory.

If the properties file is not found, the IDE offers to create a default one; click **Yes**.



If a database configuration which is different from the default for RelL (postchain/postchain/postchain as database/user/password) has to be used, specify it in the created file. Then run the app again.

RelL program output and Postchain node log is shown in the Console view. To stop a node, click the *Terminate* (red square) icon in the Console view.



One can use `curl` to call a query from the running node:

```
curl http://localhost:7740/query/
↪ 0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF -X POST -d '{"type
↪ ":"q"}'
```

By default, the blockchain RID is 0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF. It can be changed in a *run configuration*.

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 allows to run multiple blockchains in one Postchain node.

Example of a minimal run.xml:

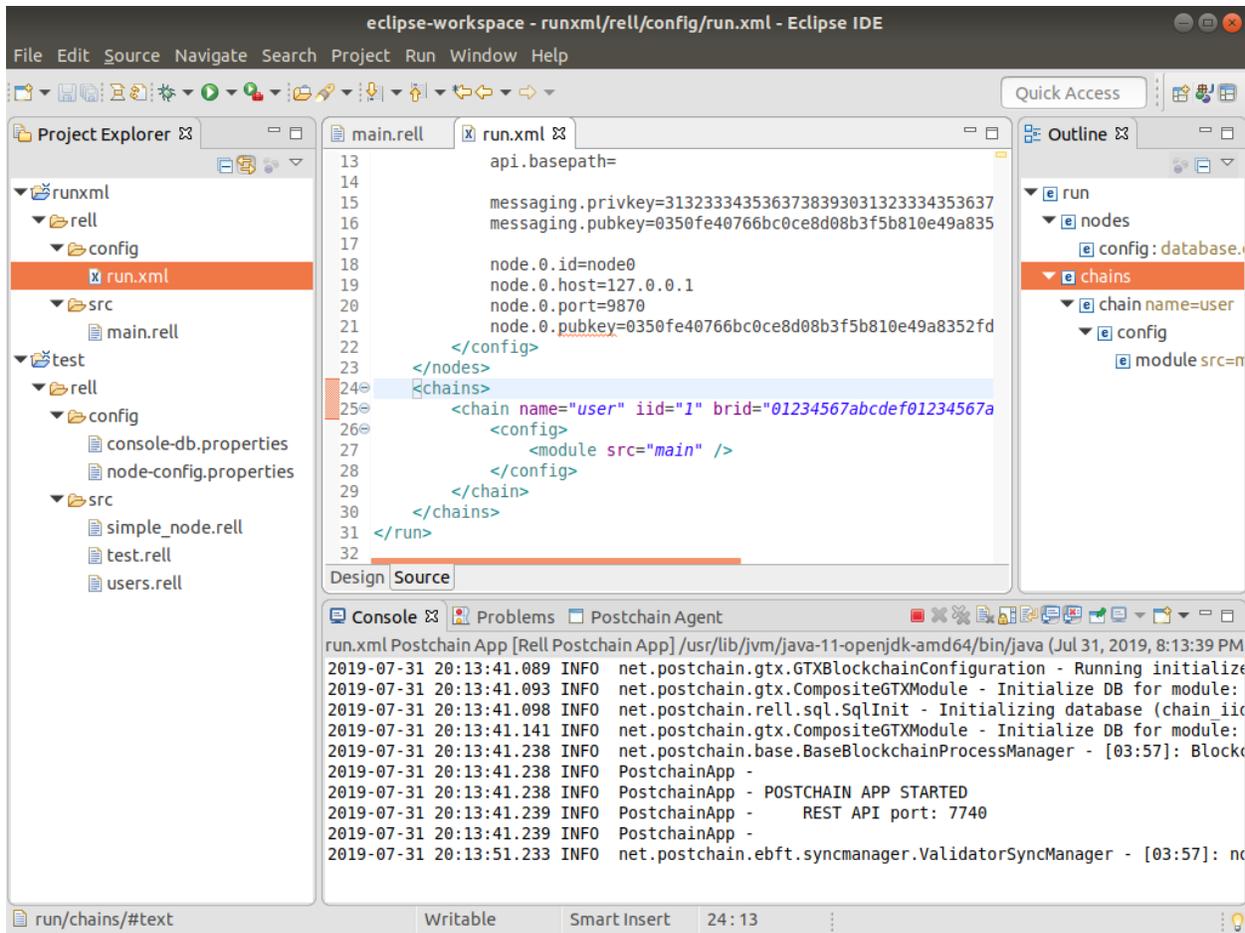
```
<run>
  <nodes>
    <config>
      database.driverclass=org.postgresql.Driver
      database.url=jdbc:postgresql://localhost/postchain
      database.username=postchain
      database.password=postchain
      database.schema=reII_app

      activechainids=1

      api.port=7740
      api.basepath=

      messaging.
      ↪privkey=3132333435363738393031323334353637383930313233343536373839303131
      messaging.
      ↪pubkey=0350fe40766bc0ce8d08b3f5b810e49a8352fdd458606bd5fafa5acdcdc8ff3f57

      node.0.id=node0
      node.0.host=127.0.0.1
      node.0.port=9870
      node.0.
      ↪pubkey=0350fe40766bc0ce8d08b3f5b810e49a8352fdd458606bd5fafa5acdcdc8ff3f57
    </config>
  </nodes>
  <chains>
    <chain name="user" iid="1" brid=
      ↪"01234567abcdef01234567abcdef01234567abcdef01234567">
      <config>
        <module src="main" />
      </config>
    </chain>
  </chains>
</run>
```



RelI Simple Postchain Test

Runs tests written in XML format, same as used by the *Web* IDE. This run option is available for XML files `*_test.xml`. For a file `X_test.xml` there must be a file called `X.rell` in the same directory. The tests defined in the XML file will be run against the RelI file. Results are printed to the Console view.

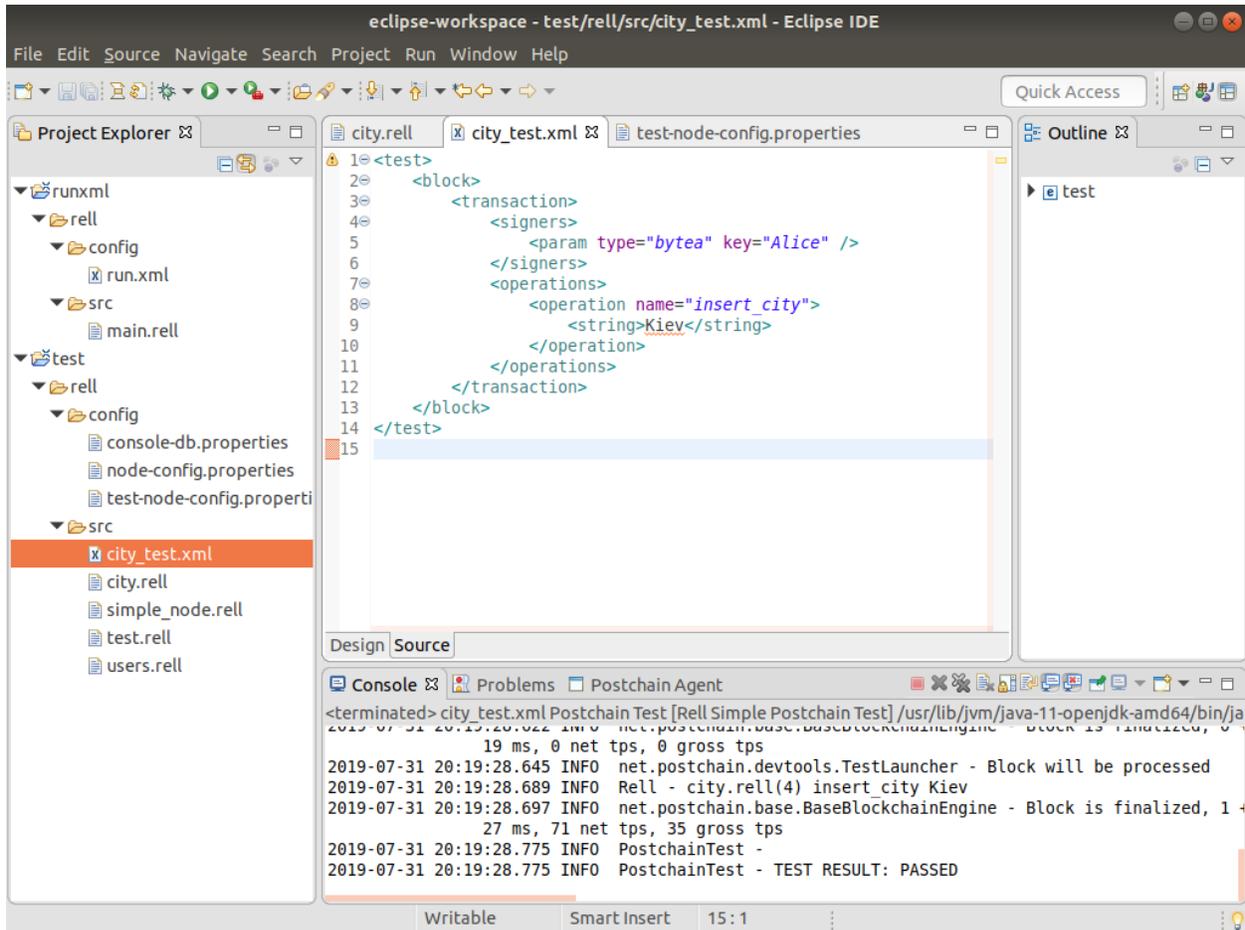
Example of a test:

```
<test>
  <block>
    <transaction>
      <signers>
        <param type="bytea" key="Alice" />
      </signers>
      <operations>
        <operation name="insert_city">
          <string>Kiev</string>
        </operation>
      </operations>
    </transaction>
  </block>
</test>
```

And a corresponding `.rell` file:

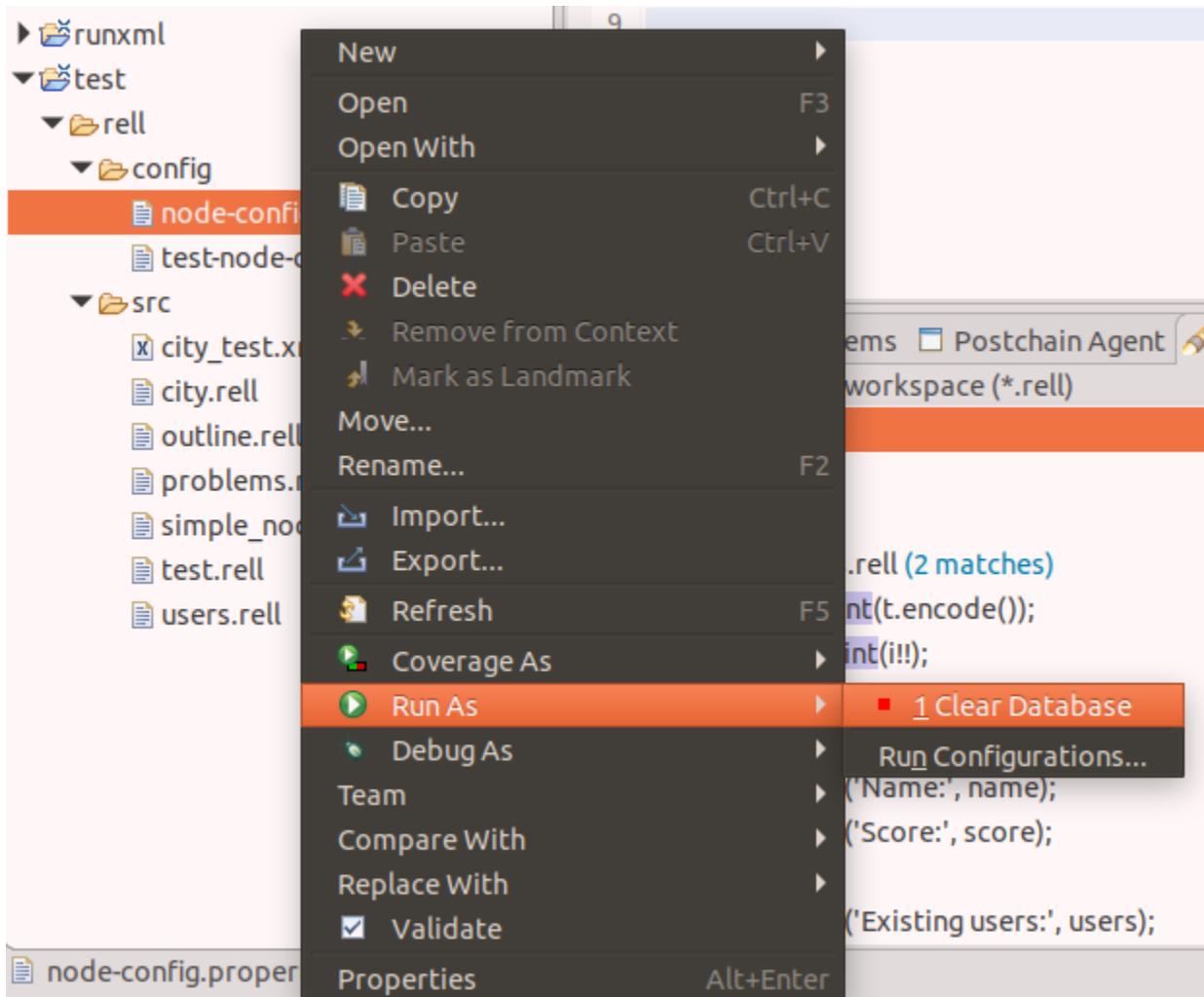
```
class city { key name; }

operation insert_city (name) {
    log('insert_city', name);
    create city (name);
}
```



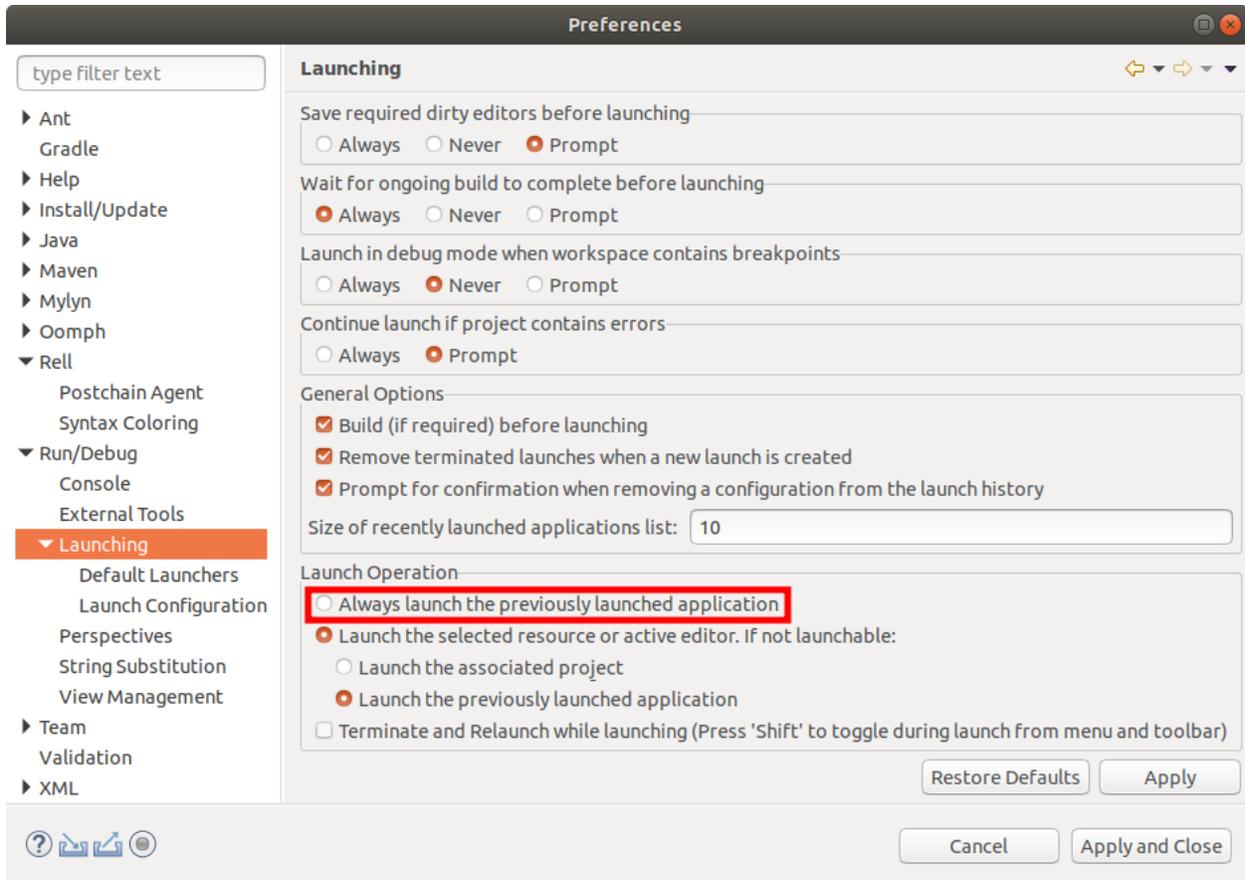
Clear Database

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

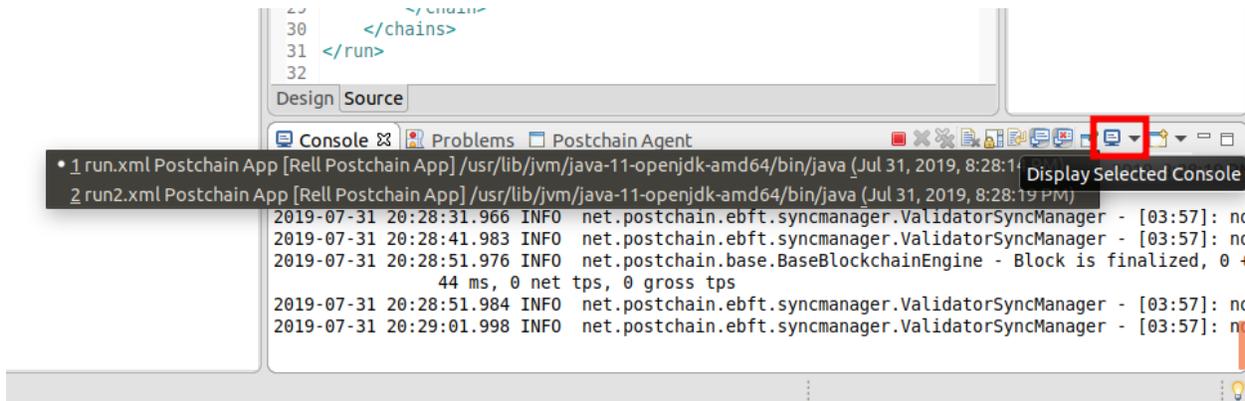
By default, CTRL-F11 (F11) shortcut runs the file of the active editor. It can be configured to run the last launched application instead, which may be more convenient, as there is no need to choose an application type. Go to the menu **Window - Preferences** (macOS: **Eclipse - Preferences**), then **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 ReII module, but different ports and database schemas.

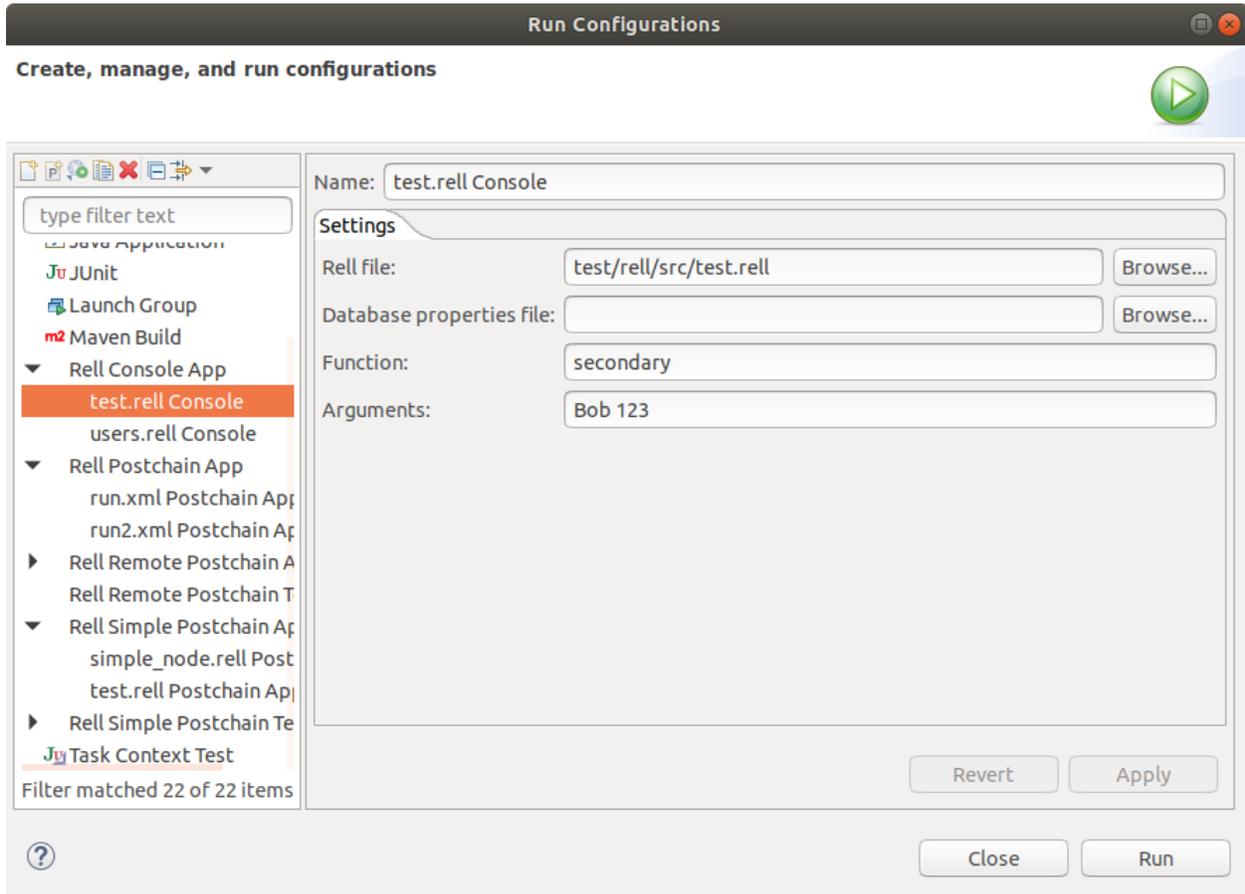
The output of all running applications will be shown in the Console view, but on different pages. It is possible to switch between the consoles of different applications using a button or a dropdown list.



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.

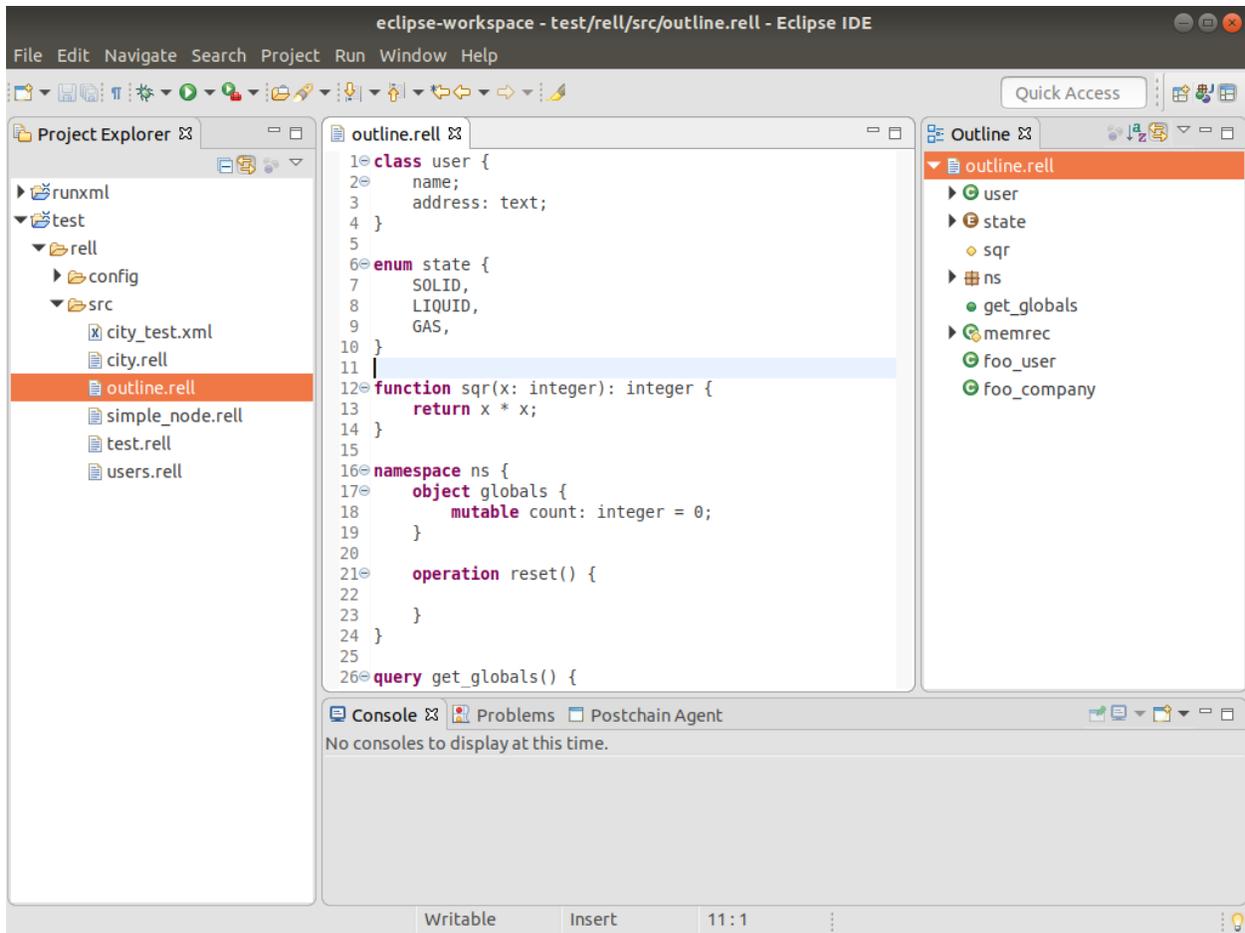
To change a run configuration, go to the menu **Run - Run Configurations...** The last launched application will be selected. Change the settings and click either **Apply** or **Run** to save the changes.



2.7.5 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.

```
    address: text;
}

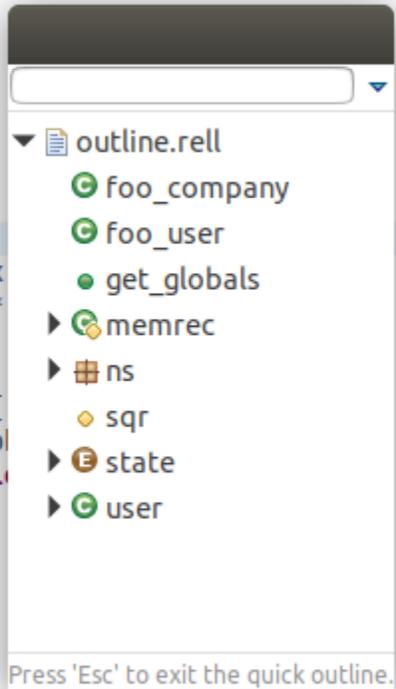
enum state {
    SOLID,
    LIQUID,
    GAS,
}

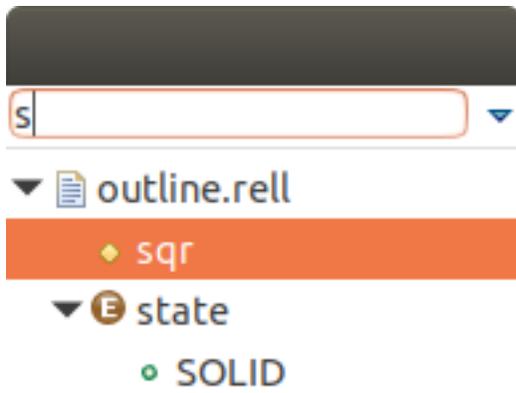
function sqr(x) {
    return x * x;
}

namespace ns {
    object global {
        mutable
    }

    operation
}

query get_globals() {
```

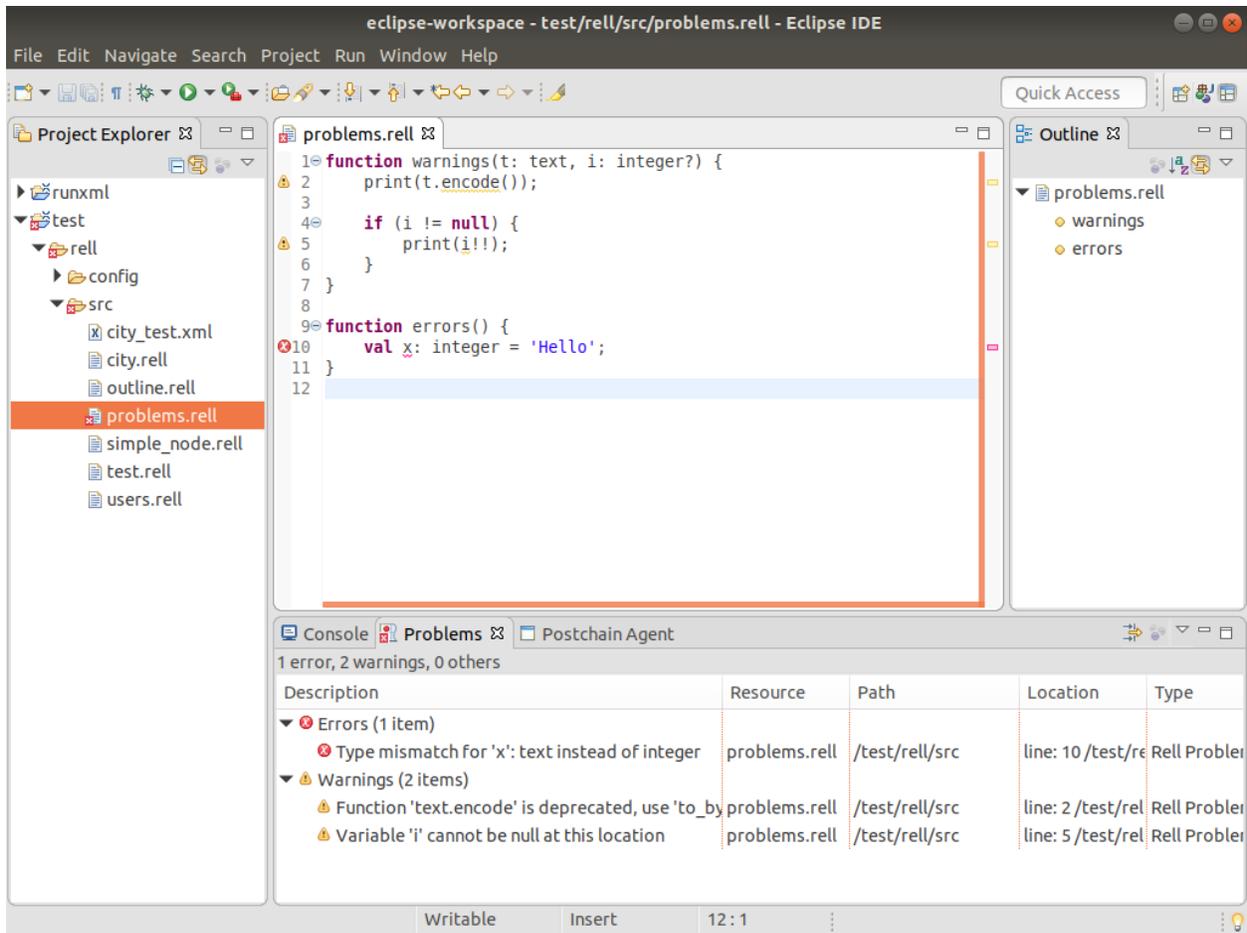




Press 'Esc' to exit the quick outline.

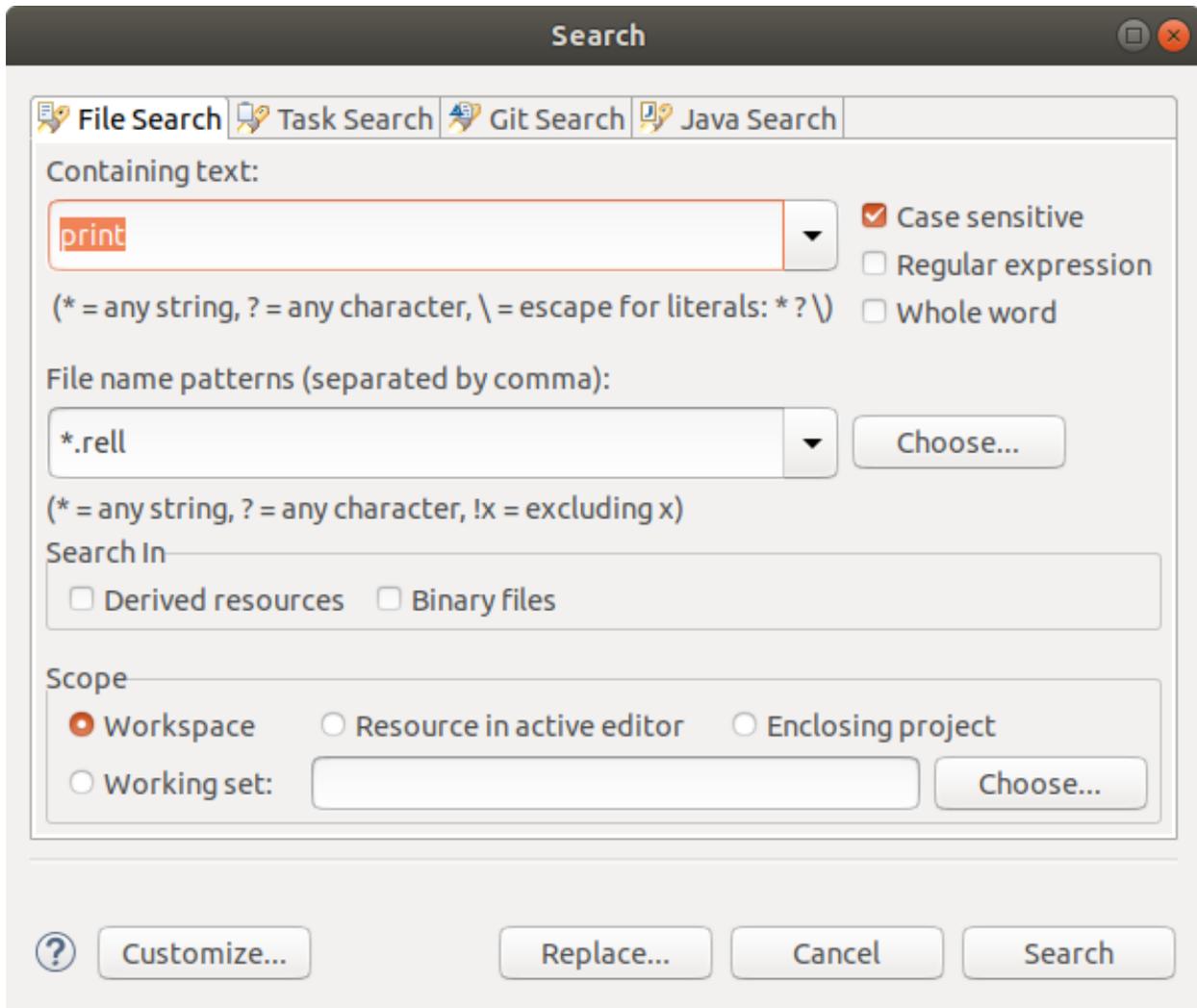
Problems View

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

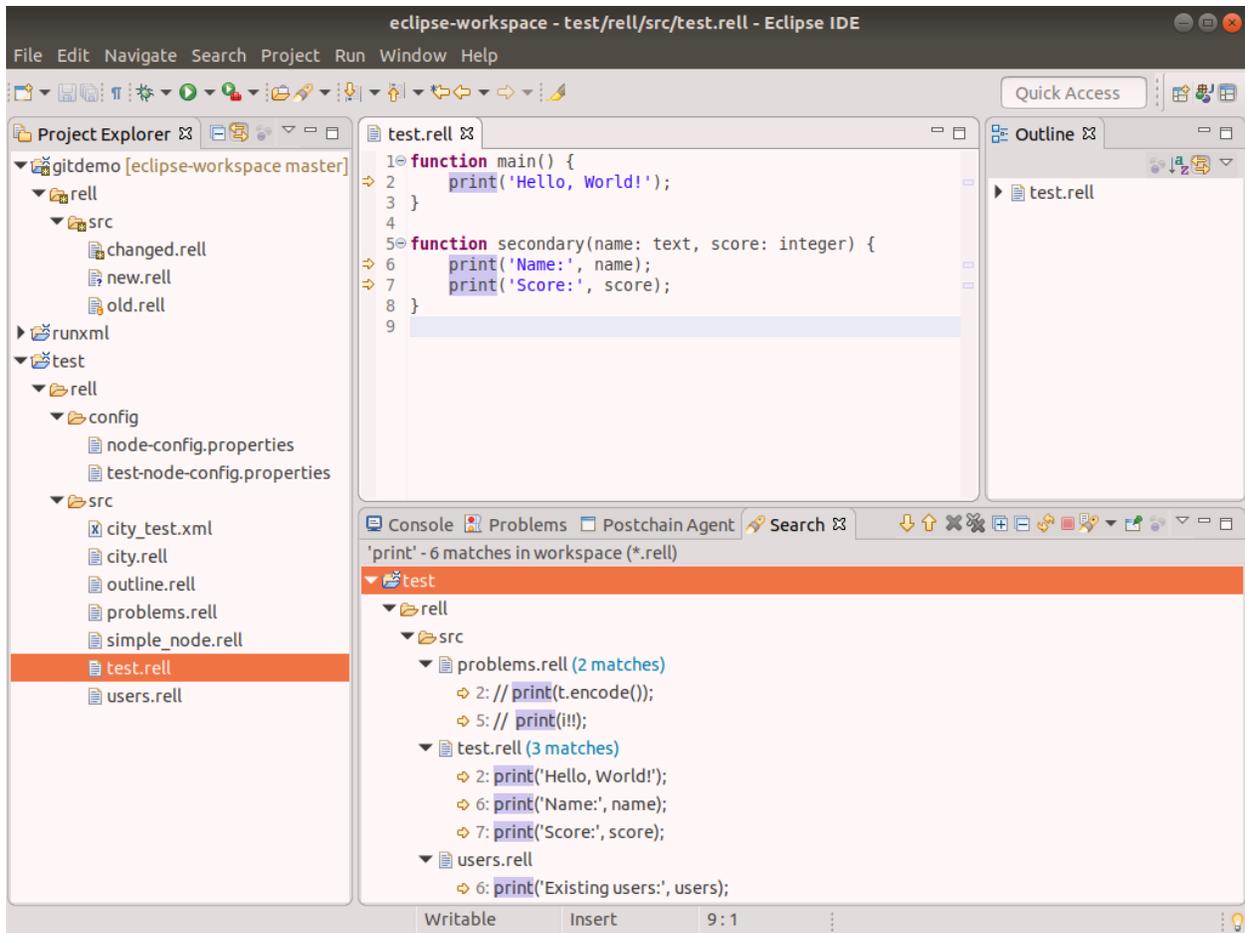


Global Text Search

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.

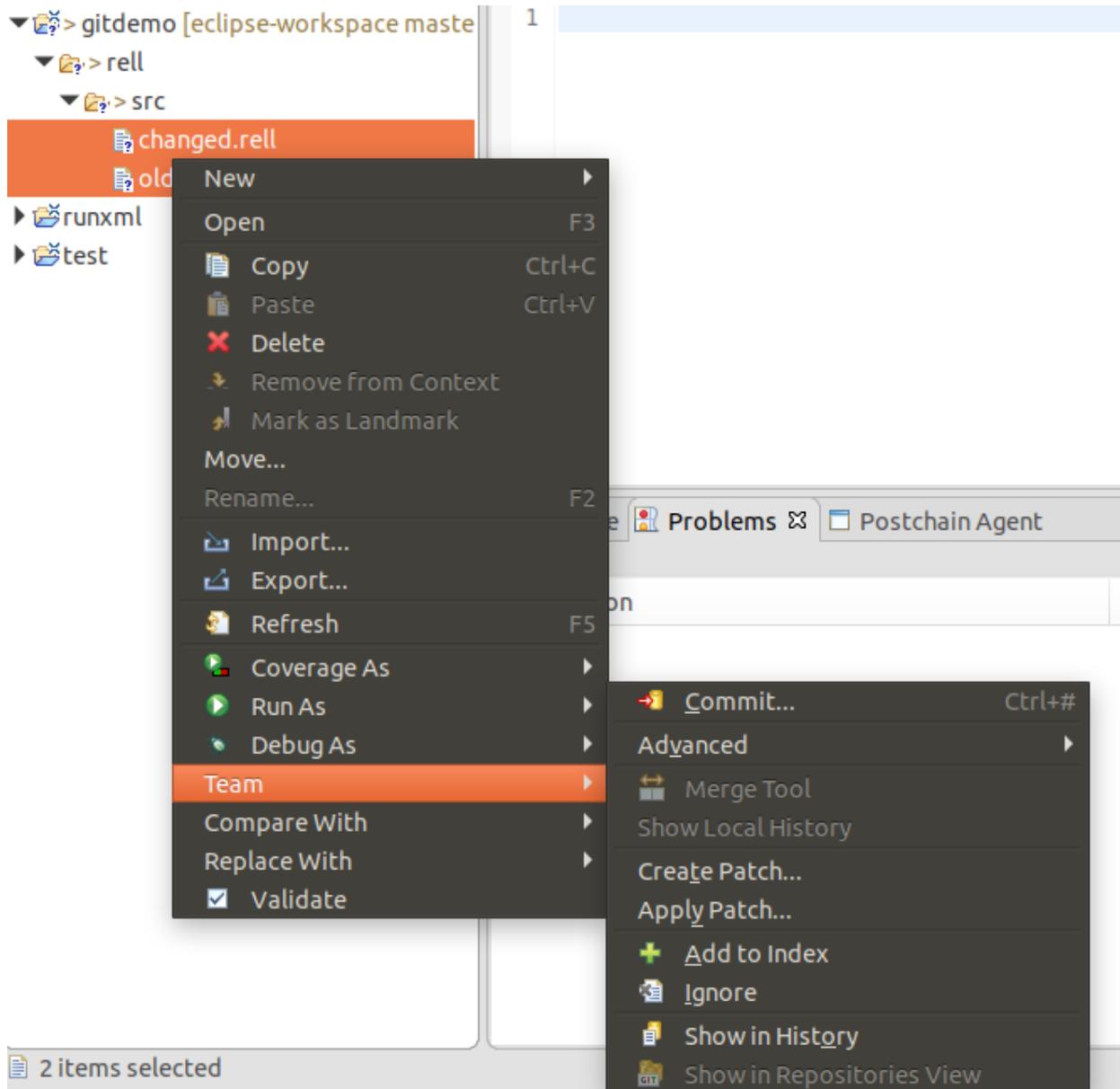


Results are shown in the Search view, as a tree.



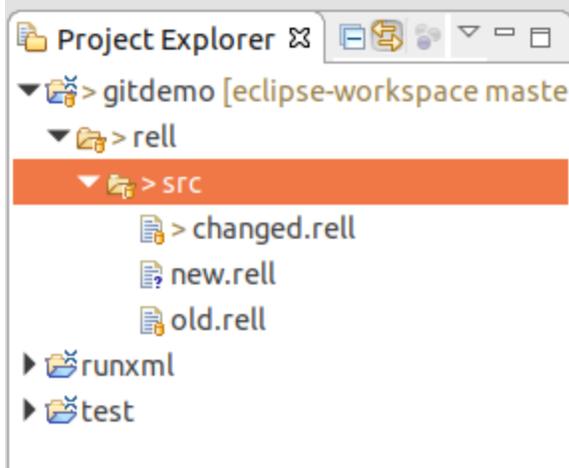
Git

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



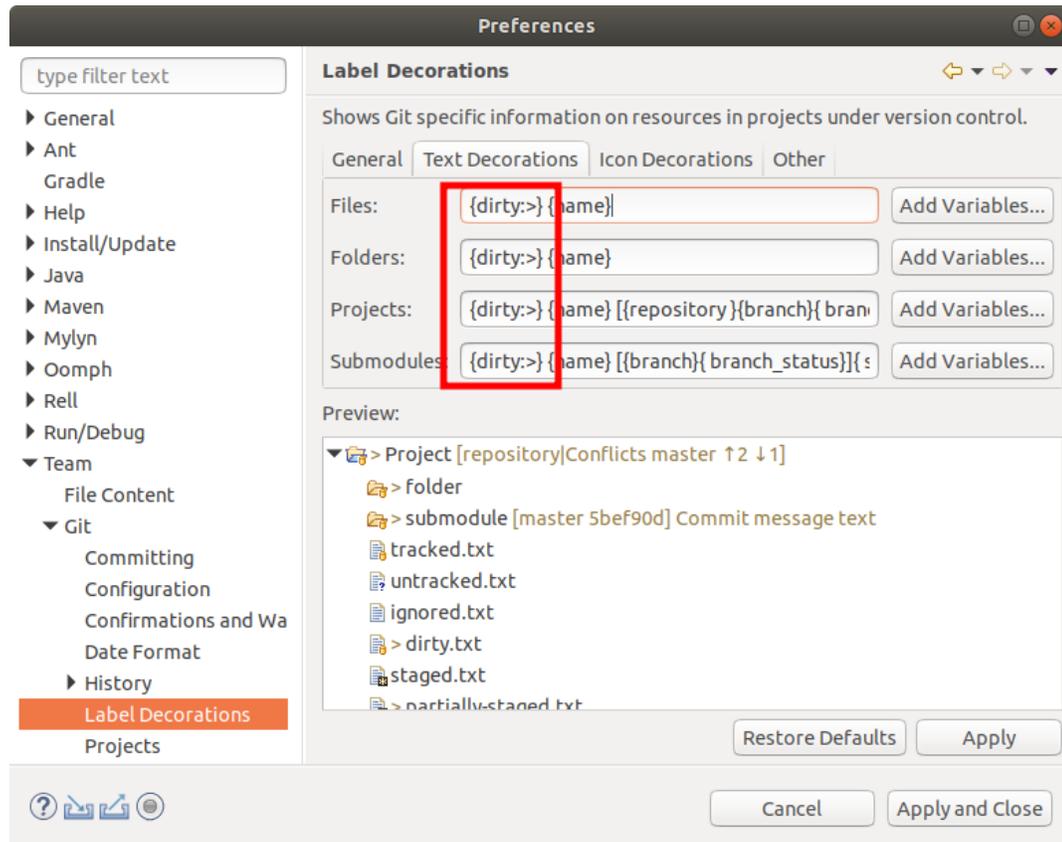
To commit file(s), click **Add to Index**, then **Commit...**

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

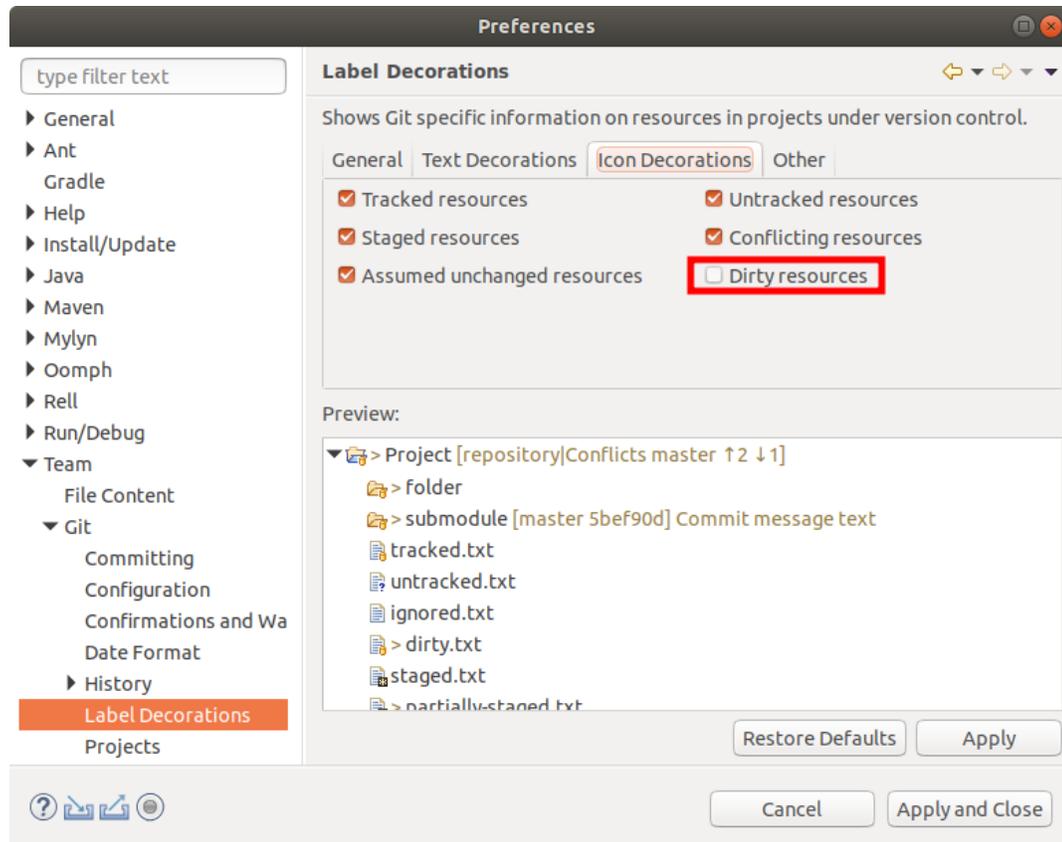


Changed files do not look very nice. To change the way how they are displayed, go to the **Window - Preferences** (macOS: **Eclipse - Preferences**) menu, then **Team - Git - Label Decorations**:

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

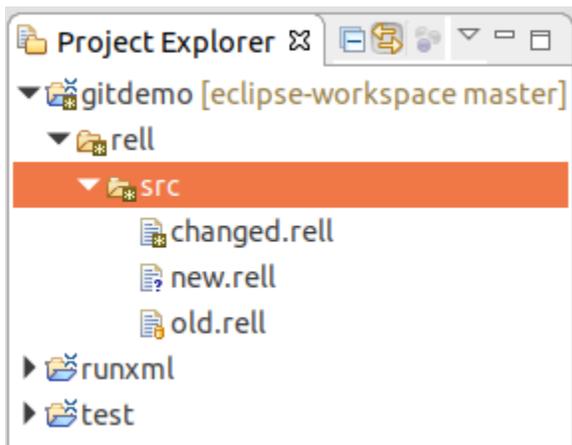


- on the **Icon Decoartions** tab: check the **Dirty resources** checkbox



- click **Apply and Close**

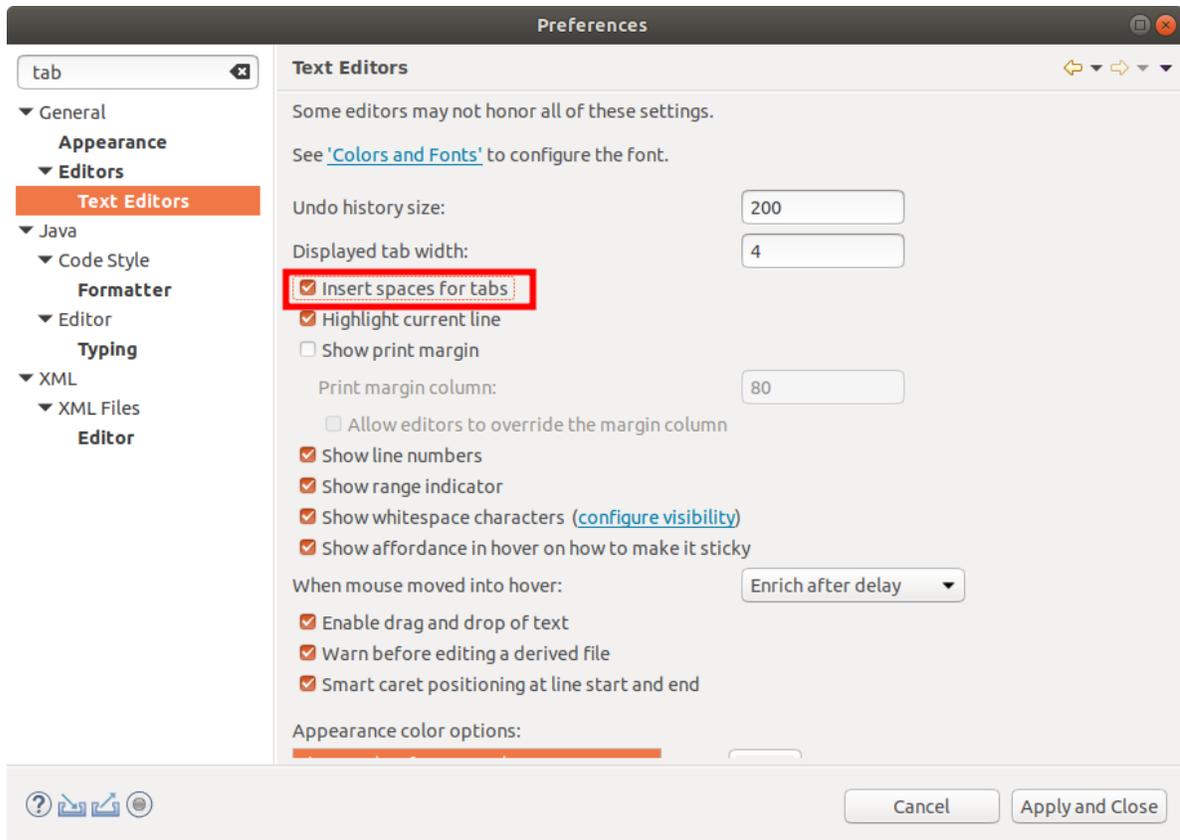
Now files look much better:



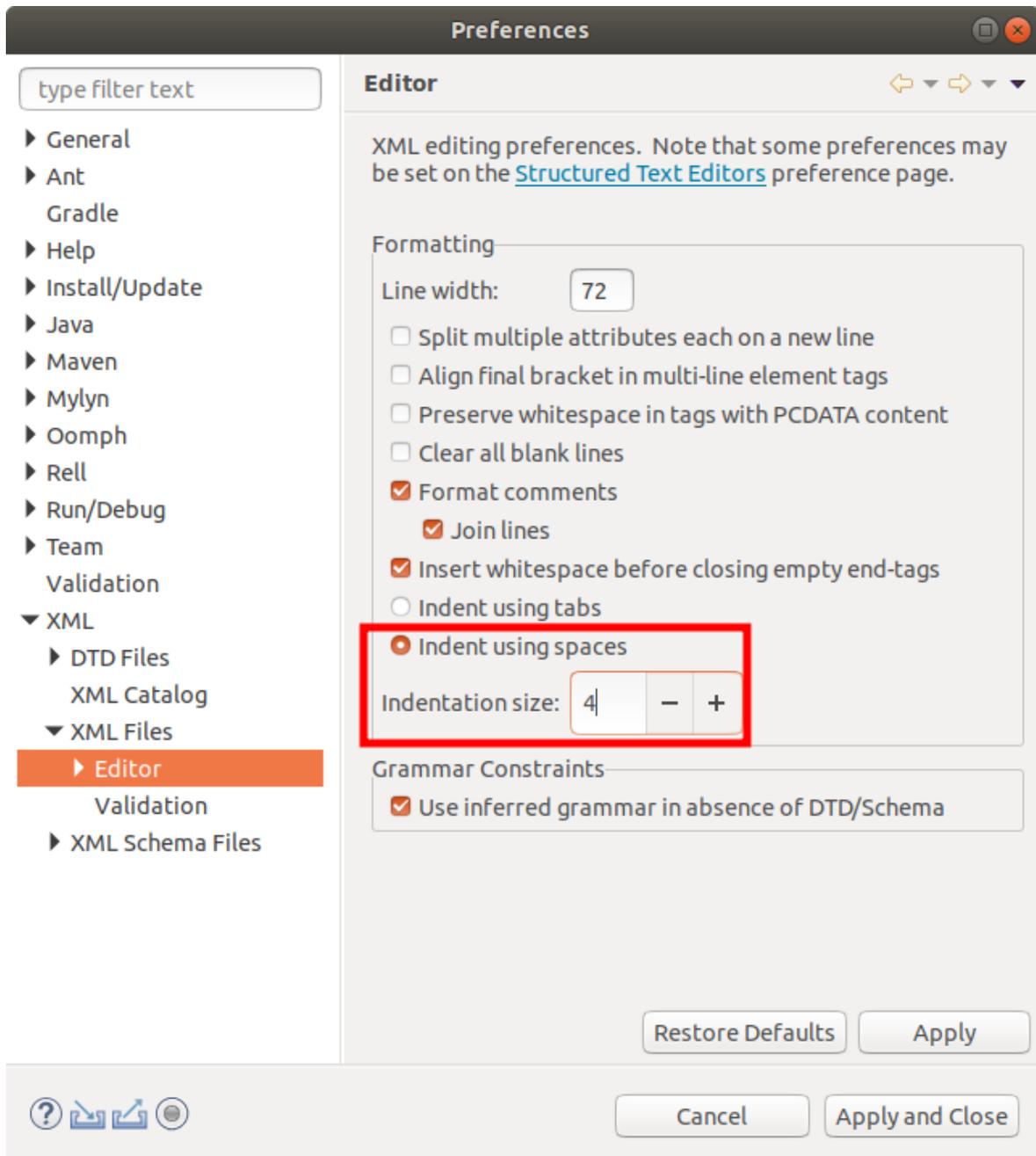
Spaces vs. Tabs

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**.

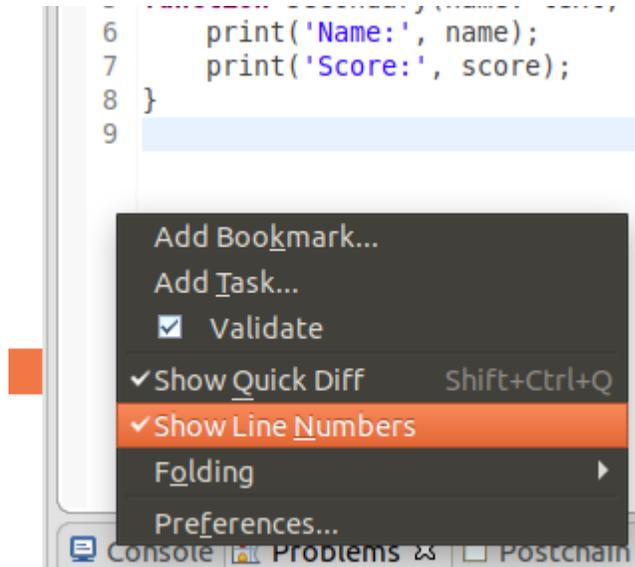


4. Go to **XML - XML Files - Editor**, select **Indent using spaces**, specify **Indentation size**: 4.

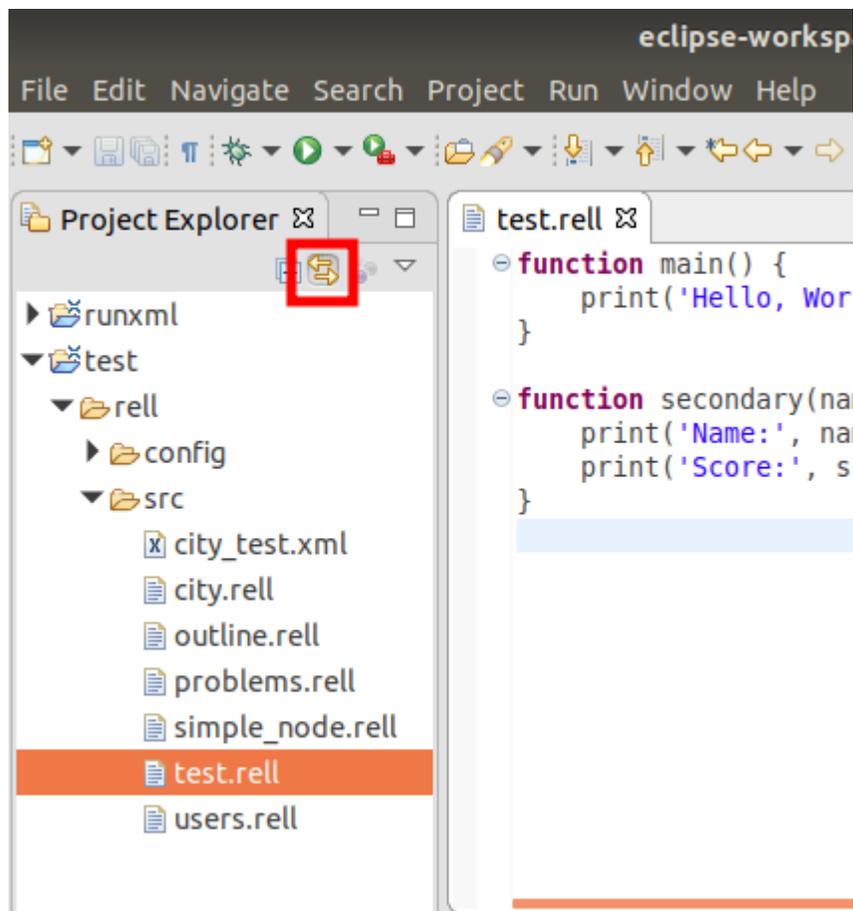


Miscellaneous

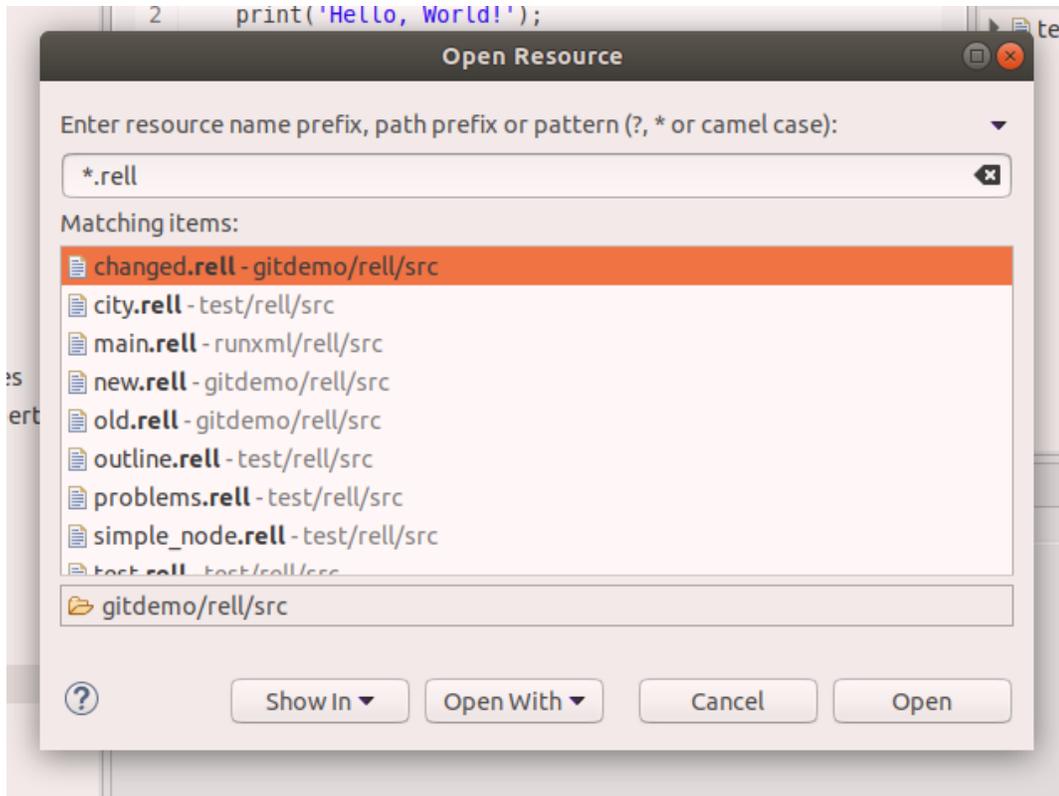
- To show or hide line numbers: right-click on the left margin of an editor, click **Show Line Numbers**.



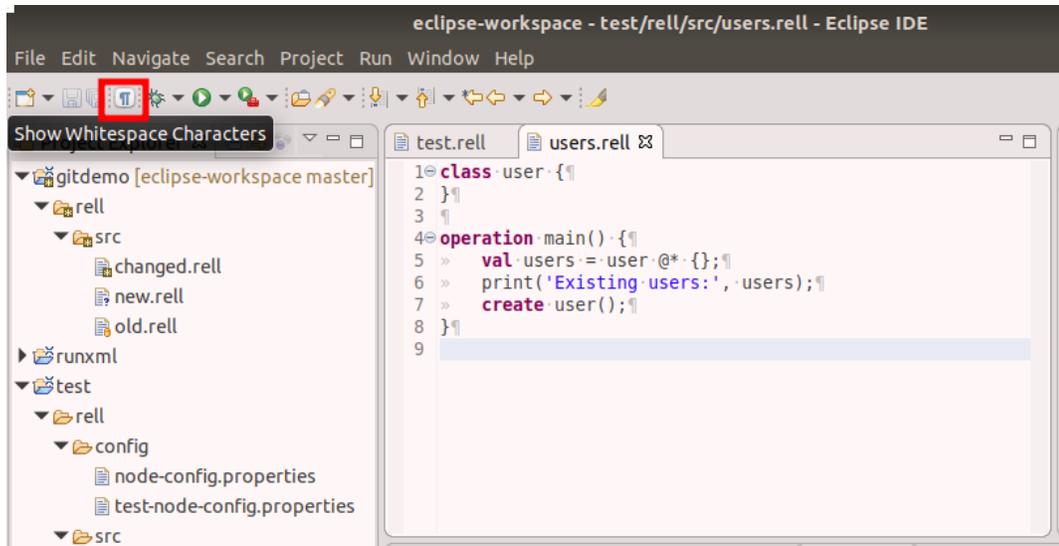
- To comment a code fragment, select it and press CTRL-/ (/).
- Activate the *Link with Editor* icon, and the IDE will automatically select a file in the Project Explorer when its editor is focused.



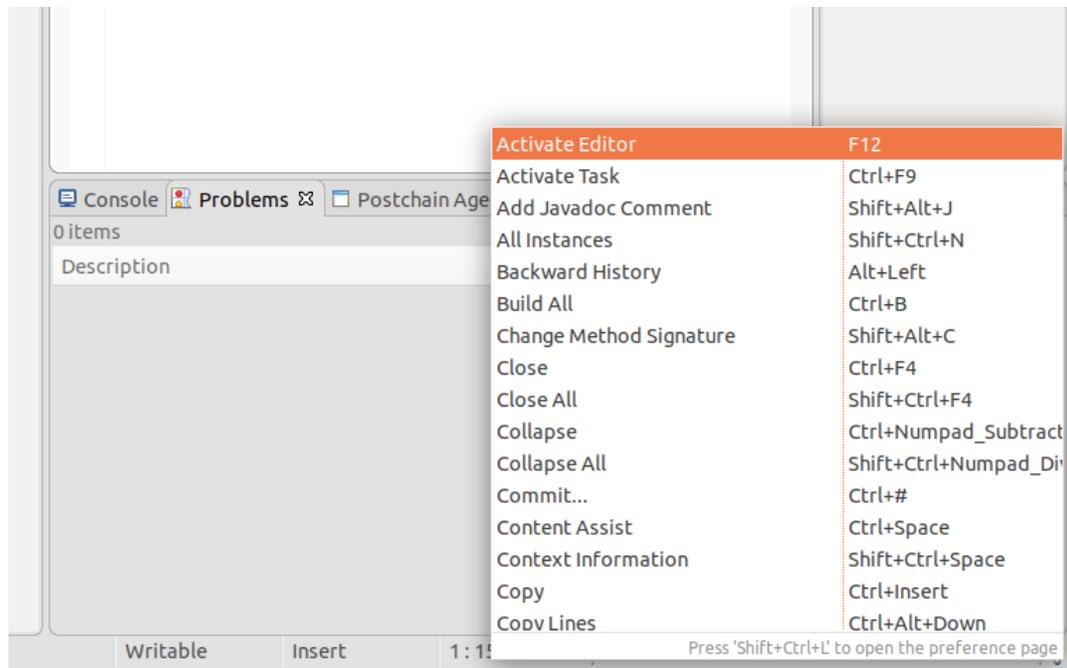
- CTRL-SHIFT-R (R) invokes the Open Resource dialog, which allows to search project files by name. Glob patterns (with * and ?) are supported.



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



- To see the full list of Eclipse keyboard shortcuts, press CTRL-SHIFT-L (L). Some shortcuts are for the Java editor, and do nothing in RelI.



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 shortcuts	CTRL-SHIFT-L	L

2.8 Run.XML

Run.XML format is used to define a run-time configuration of a ReII 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 ReII 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).

2.8.1 The Format

Example of a Run.XML file:

```

<run wipe-db="true">
  <nodes>
    <config src="config/node-config.properties" add-signers="false" />
  </nodes>
  <chains>
    <chain name="user" iid="1" brid=
↳ "01234567abcdef01234567abcdef01234567abcdef01234567abcdef01234567">
      <config height="0">
        <app module="user" />
        <gtv path="gtx/rell/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" brid=
↳ "abcdef01234567abcdef01234567abcdef01234567abcdef01234567abcdef01">
      <config height="0" add-dependencies="false">
        <app module="city" />
        <gtv path="signers">
          <array>
            <bytea>
↳ 0350fe40766bc0ce8d08b3f5b810e49a8352fdd458606bd5fafa5acdcdc8ff3f57</bytea>
          </array>
        </gtv>
      </config>
      <include src="config/city-include-1.xml" />
      <include src="config/city-include-2.xml" root="false" />
      <dependencies>
        <dependency name="user" chain="user" />
      </dependencies>
    </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 Reli 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 `gtv` 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"/>
```

2.8.2 Utilities

Those utilities are a part of the RelL language.

multirun.sh

Runs an application described by a Run.XML configuration.

```
Usage: ReIIRunConfigLaunch [-d=SOURCE_DIR] RUN_CONFIG
Launch a run config
    RUN_CONFIG    Run config file
-d, --source-dir=SOURCE_DIR
                    ReII source code directory (default: current directory)
```

multigen.sh

Creates a Postchain blockchain XML configuration from a Run.XML configuration.

```
Usage: ReIIRunConfigGen [--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
                    ReII source code directory (default: current directory)
-o, --output-dir=OUTPUT_DIR
                    Output directory
```

Example of a generated directory tree:

```
out/
├── blockchains
│   ├── 1
│   │   ├── 0.xml
│   │   ├── 1000.xml
│   │   └── brid.txt
│   └── 2
│       ├── 0.xml
│       ├── 1000.xml
│       ├── 2000.xml
│       ├── 3000.xml
│       └── brid.txt
├── node-config.properties
└── private.properties
```