	GE Capital Retail Bank 12-MONTH CD 1.05 % or \$15,000 or more	Great Rates + Safety Build your savings with a CD.	= Peace of Mind. Get started → Member FDIC ² ^{M See ductowers}
About Subscribe RSS		TORS' BLOG TPMDC d Comments & News 1	MUCKRAKER IDEALA TPM Prime
TPM Editor's Blog			- ADVERTISEMENT -
Is This Who Runs Prism	1?		
SHARE THIS STORY ON FA	ACEBOOK TWEET THIS		
JOSH MARSHALL - JUNE 7,	2013, 7:57 AM		
 because after reading it throug little doubt that Palantir is doit those tech companies, even if the From an anonymous reader I don't see anyone out there favorite news source, so here "PRISM" is the government" technology from Palantir. Pathat's now valued at well owned to be a set of the provide the technology from Palantir. 	with this theory, and TPM is n e goes: 's name for a program that uses alantir is a Silicon Valley start-1	my own there's ent is doing with Give this a read. ny s up	
1			

"We build software that allows organizations to make sense of massive amounts of disparate data. We solve the technical problems, so they can solve the human ones. Combating terrorism. Prosecuting crimes. Fighting fraud. Eliminating waste. From Silicon Valley to your doorstep, we deploy our data fusion platforms against the hardest problems we can find, wherever we are needed most." http://www.palantir.com/what-we-do/

They're generally not public about who their clients are, but their first client was famously the CIA, who is also an early investor.

With my theory in mind, re-read the denials from the tech companies in the WSJ (emphasis mine):

Apple: "We do not provide any government agency with direct access to our servers..."

Google: "... does not have a 'back door' for the government to access private user data..."

Facebook: "... not provide any government organization with direct access to Facebook servers..."

Yahoo: "We do not provide the government with direct access to our servers, systems, or network..."

These denials could all still be technically true if the government is accessing the data through a government contractor, such as Palantir, rather than having direct access.

I just did a quick Google search of "Palantir PRISM" to see if anyone else had this theory, and the top results were these pages:

https://docs.palantir.com/metropolisdev/prismoverview.html

https://docs.palantir.com/metropolisdev/prismexamples.html

Apparently, Palantir has a software package called "Prism": "Prism is a software component that lets you quickly integrate external databases into Palantir." That sounds like exactly the tool you'd want if you were trying to find patterns in data from multiple companies.

So the obvious follow-up questions are of the "am I right?" variety, but if I am, here's what I really want to know: which Palantir clients have access to this data? Just CIA & NSA? FBI? What about municipalities, such as the NYC police department? What about the governments of other countries?

What do you think?

FWIW, I know a guy who works at Palantir. I asked him what he/they did once, and he was more secretive than my friends

TPMLIVEWIRE

- <u>All</u>
- <u>U.S.</u>
- World
- Guns
- <u>Tech</u>

McCain, Feinstein To Tour Guantanamo Bay Detention Facility

14 minutes ago

Sens. John McCain (R-AZ) and Dianne Feinstein (D-CA) are making a trip to tour the detention facility in Guantanamo Bay, Cuba on Thursday, according to the Arizona senator's Twitter account

- ADVERTISEMENT -





- ADVERTISEMENT -

AdChoices D

Late Update: Another reader notes that Bridgewater Associates LLP, one of the largest hedge funds in the world, is also a major client of Palantir, which appears to be confirmed by many press reports.

Later Update: Here's a video of Alexander Karp, CEO of Palantir, describing what the company does ...

Yet More Update: For yet more of a sense of what Palantir does for the US government, here's a hypothetical of what they make possible for counterterrorism analysts in the US intel committee, as described in a 2011 article in Bloomberg ... Metropolis DevZone > Data Integrator's Guide > Prism Overview

Prism Overview

Prism is a software component that lets you quickly integrate external databases into Palantir. Specifically, it lets you build high-performance Data Engine based providers without writing any code. Instead, you define simple configuration files and then Palantir automatically constructs the data provider and database code for you. This ensures that all data access goes through well tested, high-performance code paths. Also, you can iterate more quickly because you can modify and reload Prism-based data providers without restarting the server.

This page covers the following topics:

- Is Prism Right for My Data?
 - Prism vs Web API
- Creating Your First Prism Provider
 - Prerequisites
 - Turn on Prism
 - Create a config file
 - . Load the Prism Provider
 - Add Timeseries Data
- Previous Versions
 - Version 3.18
 - Version 3.16
 - Version 3.15
 - Version 3.14
- See Also

Is Prism Right for My Data?

Prism supports the following data sources:

- MS SQL Server
- PostgreSQL
- Oracle
- Netezza
- CSV (Comma-separated values) files*

Prism works by mapping the columns in your database into models and metrics in the system. It works well when your data set has an obvious column or columns to use as the model identifier (usually the primary key). We'll see examples of good data sets later.

Note that the use of CSV files is meant for proof of concept type of applications *only* and will cause performance problems if used at production level scale. At that level of scale you should use a database such as Postgres. For examples of how to use Postgres with Prism, see Prism Examples.

Prism vs Web API

Prism is different than the Web API Data Provider and the Importer application because it connects your database directly to Palantir. You don't store a separate copy of the data in the Web API database. Thus, you don't need to fit your data into the Web API's database format. Instead, your Prism configuration file contains mappings from your external data source to the models and metrics you want to see in Palantir.

Creating Your First Prism Provider

Let's create a Prism data provider that exposes some stocks in a CSV file. When we're done, you'll be able to access the CSV data from applications like Explorer and Chart.

Prerequisites

You should know how to access the file system on the Palantir dispatch server (usually through secure shell).

Turn on Prism

Open the file **config.properties** and set **enabled** to **true**. This file is located in the Prism folder on your dispatch server (typically /opt/palantir/<server>/providers/prism/. It looks like this:

```
# provider-specific properties for Prism
pf.provider.prism.enabled=true
```

There are many config.properties files for different components on the server. Make sure you edit the file in the Prism folder. For more information, see Property Files

Now that Prism is on, the server creates a data provider for every Prism configuration file in the <server>/providers/prism folder. So, let's create our first Prism config file.

Create a config file

- Create a file called MyFirstProvider.carbon in the Prism folder (<server>/providers/prism).
 Files that end with *.carbon are called Prism configuration files. The provider's name matches the filename, so it will be MyFirstProvider.
- 2. Copy and paste the following settings into the file.

```
import com.palantir.finance.commons.data.model.*
Config {
    Csv {
        rootDirectory = './providers/prism'
        nullValues = ['null', 'empty', 'n/a']
    }
}
CommonStock{
    File {
        filename = 'MyProperties.csv'
        modelSource = true
        id = [tokens: ['symbol'], columns:['model']]
        properties << ['name','description']</pre>
```

}

Prism config files contain **Carbon script**, which is a language Palantir created specifically for Prism. Its syntax is based on Groovy, but don't worry if you aren't familiar with Groovy; we'll cover all you need to know in later examples. Let's see what this file is doing:

- The **Config** block contains info about your data source. Usually, it contains database information like db type, url, and password, but because we are connecting to a CSV data source, it contains the location and format of the CSV files.
- The **CommonStock** block maps the data in the CSV file to models and metrics in the system. In this case, it says that the data in MyProperties.csv should be put into models of type CommonStock. Also, it says to map the CSV columns model, name, and description to the properties symbol, name, and description respectively.
- The **import** line lets you specify model types in the system without having to type their full name. For example, you get to type CommonStock instead of com.palantir.finance.commons.data.model.CommonStock.
- 3. Put MyProperties.csv in your Prism directory. This directory must match the rootDirectory setting in your config file.

Load the Prism Provider

At this point, we have defined our data provider but the Palantir server has not loaded it. There are two ways to load the new data provider:

- restart the dispatch server either from Palantir Configuration Server (PCS) (if installed) or from the command line by running one of the scripts in <server>/bin.
- run the MBean operation com.palantir.finance.data.PrismProvider.addPrismProvider (available in version 3.16 and later). To access the MBean, you must connect to the server via JMX (usually with JConsole). For more information, see Monitoring and Managing Servers with JMX.

After loading the provider, you should be able to see your data. Here what it looks like in the $\boxed{\mathbb{V}}$ Explorer application.

] C - Q]	New Explorer Gro	up* - Explorer Q Palantir	- 0
Home Export	Category Filter			
lect starting universe: Ni models Explorer Type	Associations country Provider countryCod baseCurrency countryCod company Category Filters	and a second	Percentile Percentile Relative Position Other Filters	Group
Enter an instrument, me	tric, or group expression.			Home Syz Variable
Filters Start with all	models		C Refresh Data	Expand All Expand All
Intersect Property	ValueBucketingMetric ANY of [MyFirstP	rovider]	0% .5313> 5 models	Move: Up / Down 👻 🗙
Category Metric: f _x Name = common	dataProviders Search: % of Int	Add column	average of f_X [Enter a me	tric] on or before as-of date
Name 🛋	% of In 3: ne	Check your w provider 0.1% 5 0.1% 3 0.0% 1		4: Add your properties
Name a common dataRouter imports MyFirstProvide truths webapi	er er	Check your w provider 0.1% 5 0.1% 3		4: Add your properties
Name A common dataRouter imports MyFirstProvide truths	er er	Check your w provider 0.1% 5 0.1% 3 0.0% 1		4: Add your properties
Name = common dataRouter imports MyFirstProvide truths webapi	er dels S	Check your w provider 0.1% 5 0.1% 3 0.0% 1	1 💽 🛃 Pop out	4: Add your properties 5
Name A common dataRouter imports MyFirstProvide truths webapi Explorer Group - 5 mc Name Name for Stock 2 Name for Stock 3	er state of in 3: ne odels s Token v	Check your w provider 0.1% 5 0.1% 3 0.0% 1 Search: description description description for stock 2 best stock ever. Do not dump	I 💽 🛃 Pop out 🧃 symbol	4: Add your properties description
Name A common dataRouter imports MyFirstProvide truths webapi Explorer Group - 5 mc Name Name for Stock 2	er stock2	Check your w provider 0.1% 5 0.1% 3 0.0% 1 Search: description description for stock 2	I I Pop out I symbol STOCK2	4: Add your properties 5 & description dataProviders() 1 values: MyFirstProvider

Notice that there is a model for every row in the CSV file and also a model for the data provider itself. Also, we can see the model's symbol, name, and description.

Add Timeseries Data

- Download and put MyTimeseries.csv in your Prism directory. This file contains closing price and volume data for STOCK1 and STOCK2.
- 2. Modify the CommonStock block of your config file to look like this:

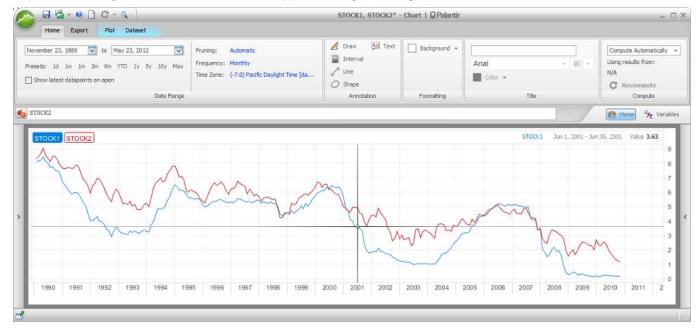
```
CommonStock{
   File {
      filename = 'MyProperties.csv'
      modelSource = true
      id = [tokens: ['symbol'], columns:['model']]
      properties << ['name','description']
   }
   File
   {
      filename = 'MyTimeseries.csv'
      id = [columns:['model']]
      Timestamp {
           dateColumn = Column(name: 'date', dateFormat: FLEXDATE)
           metrics << 'close' << 'volume'</pre>
```

```
}
```

}

Here's what is going on:

- This code **adds a second File block** for our second CSV file. Unlike the previous File block, this block is not a modelSource. This mean that we're not creating models from the data in this file. Instead, we're adding the data to models created in the other block. The models in this CSV file are mapped to the models in the CSV file by their id, which in this case, is the column model.
- The **Timestamp** block says that our CSV file has properties or metrics that change over time. The dateColumn variable defines the name and format of the column that contains dates.
- We define **metrics** close and volume in the Timestamp block because they have different values at different times.
- 3. Run the MBean operation com.palantir.finance.data.<your_prism_provider_name>.reloadProvider. This tells Prism to reload the configuration file and read the new CSV data. Note that the reloadProvider MBean will only pick up changes in the configuration file; if you alter the data only in the CSV file, you will need to restart your server to see the effects.
 - If you made changes to the data, you may need to clear Palantir's data cache to see the changes from the client. To clear the cache, run the MBean operation com.palantir.finance.service.Caches.clearAll.
- 4. (Optional) View your data in the Chart application by entering STOCK1 AND STOCK2.



Previous Versions

Version 3.18

• Prism now supports streaming results read from the database out to the data platform. This reduces the memory footprint for computations based on data in Prism providers because the system no longer needs

to collect all results in memory before returning them.

Version 3.16

- You can now add documentation (descriptions, examples, tags) to Prism properties and metrics. This documentation appears in the metric flyout window. For more information, see Carbon Reference for Prism#Documentation and Prism Examples#Parameters and Documentation.
- You can now add Prism providers without restarting the dispatch server by running the MBean com.palantir.finance.data.PrismProvider.addPrismProvider.
- You can now return models in ObjectTimeSeries. In previous versions, ObjectTimeSeries could not contain models. For an example, see Prism Examples#Object Time Series Models.

Version 3.15

- You can now create metrics that return objects. In previous version, metrics must return TimeSeries a set of time/number ticks. Now, metrics can also return ObjectTimeSeries a set of time/object ticks. For an example, see Prism Examples#Object Time Series Strings.
- Prism now supports providers that expose only CSV data. In previous versions, the database configuration block was required.
- For Prism providers, the default value for the Table threshold setting has been increased to .5 (50%). In previous versions, the default value was .2 (20%). For more information, see

Version 3.14

In version 3.14, Prism providers must connect to a SQL database. This means every provider must have a valid Database block in the config section. In versions 3.15 and later, you can omit the Database block if your provider exposes only CSV data. For more information, see Carbon Reference for Prism#Database.

See Also

Carbon Reference for Prism Prism Examples

Need Help? Email us at metropolis-support@palantir.com © 2013 Palantir Technologies · Terms of Use · Privacy and Security Statement **Metropolis DevZone** > Data Integrator's Guide > Prism Overview > Prism Examples

Prism Examples

Prism lets you quickly import external databases into Palantir by creating data providers. Each section of this page includes a Carbon and a PostgreSQL script that creates a working data provider.

Prerequisites

Before you begin, you should read the Prism Overview and be familiar with the Carbon Reference for Prism. You should also be able to run arbitrary scripts on a PostgreSQL database.

No.	Name	Topics
1	Connecting to Databases	 Database connection
2	Models, Metrics, and Properties	 Model sources Metrics Properties TimeSeries data
3	Updating Data Using Events and Triggers	 Units Variables Events Triggers Filters
4	Parameters and Documentation	Metrics with parametersMetric documentation
5	Column Transforms and Closures	Imported Java typesClosuresOutput type
6	Advanced Properties	Multi-valued propertiesTime-sensitive propertiesNull values in database
7	Historical Membership	Multi-valued time-sensitive propertiesModel types as property outputs
8	Resolving Models	 Resolution properties
9	Object Time Series - Strings	. Import object time series data

10	Object Time Series - Models	 Import object time series data

Connecting to Databases

The Carbon script included in this example logs into a database and exposes it as a data provider in Palantir. The script won't add any data; it will only ensure that you can connect to the database.

1. Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

Learning Tip

Read the comments in the provided Carbon script to understand how the script works.

```
import com.palantir.finance.commons.data.model.*
Config {
    Database {
        // The format for driver and url string are derived from your database
        // system. Check the Carbon Reference for Prism page in the
        // documentation to see valid strings for different database systems.
        driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
        username = "myUsername"
        password = "myPassword"
    }
}
```

 (Re)start your Palantir instance's dispatch server and client. Open Explorer, change the starting universe to all models and add a Provider filter. Your data provider (whose name is the same as the name of your Carbon script file) will appear in the list of providers. Now you're ready to start importing models, metrics, and properties from your database.

Jau	gory Metric: 🎊 dataProviders Search: Manually add element: 🏟		
	Name A	% of Input	Size
Π	common	99.9%	
	dataRouter	0.0%	1
	imports	0.0%	1
~	myExampleProvider	0.0%	1
	truths	0.1%	3
	webapi	0.0%	1

Models, Metrics, and Properties

The scripts in this example add the stocks MYFIRSTSTOCK and MYSECONDSTOCK to Palantir, as well as metrics that return each stock's name, symbol, trading country, and closing price time series.

- 1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data.
- 2. Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your

database's login information.

```
import com.palantir.finance.commons.data.model.*
Config {
    Database {
        driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
        username = "myUsername"
        password = "myPassword"
    }
}
CommonStock {
    "models" {
        modelSource = true
        // Creates the model-id. Taken together, The values in 'columns'
        // uniquely identify each model.
        id = [columns: ["id"], tokens: ["symbol"]]
        // For instance, calling myCommonStock.country() returns the value
        // in the countrytradingin column.
        properties << [id: "name", countrytradingin: "country"]</pre>
    }
    "timeseries" {
        // There should be a foreign key constraint on timeseries.id that
        // references models.id. If there isn't, and timeseries.id contains a value
        // that isn't in in models.id (case-sensitive), you won't be able to access
        // your timeseries data.
        id = [columns: ["id"]]
        Timestamp {
            // For instance, calling myGenericInstrument.close()
            // returns a TimeSeries made from the 'date' and 'metricl' columns
            dateColumn = Column(name: "date", dateFormat: FULLDATE)
            metrics << [ metric1: "close"]</pre>
        }
    }
}
```

Performance Tip

Palantir builds a set of models from the model source's id columns using a SQL 'SELECT DISTINCT' clause. For that reason, your id columns should be as unique as possible (ideally, using a SQL 'UNIQUE' constraint) to maximize performance.

3. (Re)start your Palantir instance's dispatch server and client. Open Calculator and enter MYFIRSTSTOCK.close() (whose final value is 90), MYFIRSTSTOCK.name() ("myFirstStock"), and MYFIRSTSTOCK.country() ("United States"). Now you've successfully imported your first models, metrics, and properties.

Cal	alculation Panel	
6		
3:	MYFIRSTSTOCK.countr ans4 = Aa United States	уO
2:	MYFIRSTSTOCK.name() ans3 = <u>Aa</u> myFirstStock	
1:	MYFIRSTSTOCK.close()
	ans1 = M	90 (2012-07-17)

Updating Data Using Events and Triggers

The scripts in this example import the stocks MYFIRSTSTOCK and MYSECONDSTOCK, as well as data for MYFIRSTSTOCK's closing price. It also adds events and triggers so that Palantir automatically reloads data from the data provider at regular intervals. Additionally, since we want to work with a subset of the rows in the database we'll use a filter to select only the rows where that condition is true (similar to a SQL 'WHERE' clause).

1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data. The table below is a sample of the data in the "timeseries" table.

id	date	tsvalue1	units
myFirstStock	2012-07-19	100	USD/share
myFirstStock	2012-07-19	200	GBP/share
myFirstStock	2012-07-18	100	USD/share
myFirstStock	2012-07-18	200	GBP/share

2. Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir (for example, CustomStocks.carbon). Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

```
import com.palantir.finance.commons.data.model.*
Config {
   Database {
        driver = "org.postgresql.Driver"
       url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
       username = "myUsername"
        password = "myPassword"
    }
}
// Each event updates the data for the given entity in the given scope. Here,
// the first event will update the data for each PROPERTY specified in ALL Table
// blocks in this file.
def propertyEvent = [ Event(entity: PROPERTY, scope: ALL) ]
def metricEvent = [ Event(entity: METRIC, scope: ALL) ]
def modelEvent = [ Event(entity: MODEL, scope: ALL)]
// The first trigger executes the propertyEvent at half past every minute (for
// instance, 8:00:30, 8:01:30, 8:02:30). The second trigger executes the
```

```
// metricEvent at every minute on the minute (for instance, 8:00:00, 8:01:00)
triggers = [
    Trigger(name: "propertyTrigger",
            cronTime: "30 * * * * ?",
            events: propertyEvent),
    Trigger(name: "metricTrigger",
            cronTime: "0 * * * * ?",
            events: metricEvent),
    Trigger(name: "modelTrigger",
           cronTime: "15 * * * * ?",
            events: modelEvent)
]
GenericInstrument() {
    "models" {
       modelSource = true
        id = [columns: ["id"], tokens: ["symbol", "name"]]
        properties << [countrytradingin: "country"]</pre>
    }
    "timeseries" {
        id = [columns: ["id"]]
        // This filter tag selects only rows from the timeseries table
        // for which timeseries.units = 'USD/share'. Only those rows will
        // become part of the metrics defined in the Timestamp block.
        filter = "timeseries.units = 'USD/share'"
        Timestamp {
            dateColumn = Column(name: "date", dateFormat: FULLDATE)
            // This creates a new unit by parsing the values in the
            // timeseries.units column. Remember that the filter above
            // selects only the rows where timeseries.units = 'USD/share'.
            def dollarsPerShare = ColumnUnit('units')
            metrics << [tsvalue1: Metric(token: "close", unit: dollarsPerShare)]</pre>
        }
    }
}
```

 (Re)start your Palantir instance's dispatch server and client. Open Calculator and enter MYFIRSTSTOCK.close() to get a time series created from importing the data in the database. Note that the units of the time series are "USD/share", exactly as the Carbon script defined it by using a ColumnUnit.

Also, while the server is running, update the values in "models.countrytradingin". The events and triggers reload this data from the provider automatically every minute, so within a minute of updating the database the country() metric will return these new values.

al	culation Panel	
6		
3;	MYFIRSTSTOCK.country()	
	ans8 = Aa United Kingdom	
2:	MYFIRSTSTOCK.country()	
	ans7 = Aa United States	
1:	MYFIRSTSTOCK.close()	
	ans6 = M	- 100 USD/share (2012-07-19)

Parameters and Documentation

The scripts in this example import Instruments that represent bank accounts and time series that represent those accounts' balances over time. The database defines the time series in two distinct units, so our scripts import a metric whose parameter lets you choose whether to return the time series in one unit or the other. The parameter performs the same function as a Carbon filter, except that the filter's condition is the metric's parameter (that you choose in the client).

1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data. Below is a sample of data from the database.

id	date	tsvalue1	units
12345	2012-07-17	100	USD
12345	2012-07-17	200	GBP
12345	2012-07-16	100	USD
12345	2012-07-16	200	GBP

 Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

```
import com.palantir.finance.commons.data.model.*
import com.palantir.finance.commons.data.model.Country
import com.palantir.finance.commons.service.ontology.model.*
Config {
    Database {
        driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
        username = "myUsername"
        password = "myPassword"
    }
}
// The Metric Documentation
def desc = Documentation(
    name: "Balance of Bank Account",
    examples: ["_12345.balance(\"USD\")"],
    tags: ["bank"],
    description: "Returns a time series that represents the balance of the specified bank account.")
// The Parameter
def myParameter = Parameter(
    name: "unitsParameter",
    column: "units",
    optionsFromColumn: true,
    description: "The currency.")
// The Metric
def myMetric = Metric(
    token: "balance",
    parameters: [myParameter],
    documentation: desc)
GenericInstrument() {
    acct_open {
```

```
modelSource = true
        id = [ columns: ["account_number"], tokens: ["name", "symbol"] ]
        properties << [acct_country: Property(token: "country", output: Country.class)]</pre>
    }
    acct_balance {
        id = [ columns: ["account_number"]]
        Timestamp {
            dateColumn = Column(name: "date", dateFormat: FULLDATE)
            metrics << [balance: myMetric]</pre>
        }
    }
}
Country() {
    country_table {
        modelSource = true
        id = [ columns: ["country"], tokens: ["country", "name", "symbol"]]
    }
}
```

3. (Re)start your Palantir instance's dispatch server and client. Open Calculator and enter _12345.balance("USD") (model tokens cannot begin with numbers or special characters, so Palantir automatically prefixes these tokens with the '_' character). Notice that the autocomplete window displays a list of choices for the string parameter. Each of these choices is a distinct value in the units column that Prism automatically identified when it loaded the data provider.

Refer back to the database. We used the parameter "USD", so the metric returned a time series made from the rows where the value in the units column was "USD". You can verify that the other rows are excluded by noting that the time series in Calculator is a constant value of 100.

Calculation Panel	
6)	
2: _12345.balance("0	GBP/share")
ans11 = ///	200 (2012-07-17)
1: _12345.balance("U	JSD/share")
ans10 = 🚧	100 (2012-07-17)

Also, verify that the metric documentation looks like this:

Balance of Bank Accoun	t
Description Returns a time series that represents the bal	ance of the specified bank account.
Examples _12345.balance("USD")	
Operates On: GenericInstrument	Returns: TimeSeries
Signature balance([String unitsParameter])	
Parameters (<i>default value</i>) String unitsParameter (<i>null</i>)	
Providers mark	
Tags bank	

Column Transforms and Closures

The scripts in this example import models that represent treasury yields over given periods and properties that define each treasury yield's name, description, country, and tenor. The tenor is stored in the database as a string, so the Carbon script defines a column transform (or closure) to transform the string value in the database to the Java type org.joda.time.Period (that represents a period of time). The period property then returns that Period object rather than the string.

- 1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data.
- Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

```
import com.palantir.finance.commons.data.model.*
// Because Carbon is an addition to Groovy, and Groovy is tightly coupled with Java
// libraries, you can add Java classes and packages. The Period class is such a class.
import org.joda.time.Period
Config {
   Database {
       driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
       username = "myUsername"
       password = "myPassword"
    }
}
// The closure is similar to an anonymous function. In the following closure, note
// that the method calls and syntax are from Java libraries. Methods such as
// parseInt(), substring(), and the Period constructor used are all specified in
// Java libraries.
def TenorClosure = { it -> // 'it' is the input
   final Period p;
   int number = Integer.parseInt(it.substring(1, it.length() - 1));
   if (it.toLowerCase().endsWith("y")) {
        p = new Period(number, 0, 0, 0, 0, 0, 0);
    } else {
        p = new Period(0, number, 0, 0, 0, 0, 0, 0);
   return p; // every closure should return a value
}
GenericInstrument() {
    'example_db' {
       modelSource = true
        id = [ columns: ['model'], tokens: ['symbol'] ]
       properties << 'name' << 'description' << 'country'
       // The variable we defined is a 'transform' applied to the column. Every value in that column will
        // the transform function as an input, and the function's output (not the column's values) will ma
        // property's output. Also note that the Property function defines the output type as Period.class
       // our transform returns a Period object.)
        properties << [(Column(name: 'tenor', transform: TenorClosure)): Property(token: 'tenor', output:
    }
}
```

3. (Re)start your Palantir instance's dispatch server and client. Open Calculator and enter BC_10YEAR.tenor() to see the value returned by our column transform. Enter BC_10YEAR.tenor().getType() to see that Palantir is returning Period objects rather than strings.

Calculation Panel	
6	
2:	<pre>BC_10YEAR.tenor().getType() ans13 = Period</pre>
1:	BC_10YEAR.tenor() ans12 = P10Y

Advanced Properties

The scripts in this example import one model, MYCOMPANY, as well as the properties 'name' and 'symbol'. They also add 'departments', a multi-valued property that returns a set rather than a single value, and 'ceo', a time-sensitive property that returns different single values depending on the value-on date.

- 1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data.
- 2. Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

```
import com.palantir.finance.commons.data.model.*
Config {
    Database {
        driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
        username = "myUsername"
        password = "myPassword"
    }
}
GenericInstrument {
    'companies' {
        modelSource = true
        id = [columns: ['company'], tokens: ['name', 'symbol']]
        // A multivalued property returns a set of values rather than a single value. Every value
        // in the target column ('departments') will be in the set returned by the property. Null
        // values in the database will not be added to the set.
        properties << [departments: Property(token: 'departments', multiValued: true)]</pre>
        Timestamp {
            dateColumns = [start: Column(name: 'start_date', dateFormat: FULLDATE),
                           end: Column(name: 'end_date', dateFormat: FULLDATE)]
            // A timeSensitive property still returns only a single value, but that value changes
            // as you change the valueOn date.
            properties << [ceo: Property(token: 'ceo', timeSensitive: true, output: String.class)]</pre>
        }
    }
}
```

3. (Re)start your Palantir instance's dispatch server and client. Open Calculator and enter MYCOMPANY.departments(), which returns a set of three strings. Although the database has four rows, multi-valued properties only include the values in the column that are non-null and ignore the rest. This property includes only three strings because the fourth row of the database is empty.

Also, enter MYCOMPANY.ceo(). If your value-on is set to the current date, this returns "John Doe 4". Although there is no CEO listed for the current date, Palantir's default on-or-before behavior means that the property returns the most recent

value if no current value exists. Set the value-on date to any day in 2010 and enter MYCOMPANY.ceo() again. This time, it returns "John Doe 3," because John Doe 3 was the CEO in 2010. Because the property is time-sensitive, it changes as the value-on date changes.

Calculation Panel			
6			
3:	MYCOMPANY.ceo() // January 1, 2010		
	ans 18 = Aa John Doe 3		
2:	MYCOMPANY.ceo() // Today		
	ans17 = Aa John Doe 4		
1:	MYCOMPANY.departments()		
	ans14 = Collection of 3 objects of type String		

Historical Membership

The scripts in this example import a StockMarketIndex model with a defined history of membership and a (multi-valued and time-sensitive) metric, 'members', which returns a set of the index's members. Because it is a time-sensitive property, members() returns a different set depending on the value-on date.

1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data. Below is a sample of the database. Note that INSTRUMENT_1 is a member of MYINDEX for two separate periods, so it occupies two rows.

index_instrument_code	instrument_code	start_date	end_date
MYINDEX	INSTRUMENT_1	2009-01-01	2009-10-01
MYINDEX	INSTRUMENT_2	2009-01-01	2011-07-01
MYINDEX	INSTRUMENT_3	2010-01-01	2011-07-01
MYINDEX	INSTRUMENT_1	2010-01-01	2011-07-01

1. Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

```
import com.palantir.finance.commons.data.model.*
Config {
   Database {
       driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
       username = "myUsername"
       password = "myPassword"
    }
}
// This block defines models that will make up our membership. A property can only output
// models that resolve to models provided by the same data provider, so without this block
// our historical membership would return null rather than a set of values.
GenericInstrument() {
    'members' {
        modelSource = true
        id = [columns: ['member_code'], tokens: ['name', 'symbol']]
        properties << 'description' << 'cusip' << 'countrytradingin'
    }
}
```

```
StockMarketIndex() {
    'index_members' {
        modelSource = true
        id = [columns: ['index_code'], tokens: ['name', 'symbol']]
        properties << [member_code: Property(token: 'allMembers', multiValued: true, output: GenericInstru
        Timestamp {
            dateColumns = [start: Column(name: 'start_date', dateFormat: FLEXDATE), end: Column(name: 'end
            // Note that this property is both time-sensitive and multi-valued. That means it returns a se
            // changes with the value-on date. Also note that the output type is GenericInstrument.class,
            // modelSource block from 'members' defined.
            properties << [member_code: Property(token: 'members', timeSensitive: true, multiValued: true,
            }
        }
    }
}
</pre>
```

2. (Re)start your Palantir instance's dispatch server and client. Open Calculator and enter MYINDEX.members(). If your value-on date is the current date, then this will return null because there are no members after early July 2012. Change the value-on date to 2012-01-01, then enter the same metric. Now the metric returns a collection of three instruments, because at that point in time there were three members. Change the value-on date again, this time to 2011-01-01, and enter the metric again. This time there are eight members.

Calculation Panel					
•					
3:	MYINDEX.members() // January 1, 2011 ans3 = Collection of 8 objects of type GenericInstrument				
2:	<pre>MYINDEX.members() // January 1, 2012 ans2 = Collection of 3 objects of type GenericInstrument</pre>				
1:	MYINDEX.members() // Today ans1= null				

Open this collection of eight instruments in Table, then add the cusip() metric to the table as a column. Notice that the value returned is the same value as in the "members" table. Even though the members of the index were specified in the "index_members" table, they resolved to models imported from the "members" table. For more information about model resolution, see Resolving Models (below) and Model Overview.

Name	cusip() ≏
1 R INSTRUMENT_1	INSTRUMENT_1_cusip
2 👷 INSTRUMENT_2	INSTRUMENT_2_cusip
3 👷 INSTRUMENT_3	INSTRUMENT_3_cusip
4 R INSTRUMENT_4	INSTRUMENT_4_cusip
5 R INSTRUMENT_5	INSTRUMENT_5_cusip
6 👷 INSTRUMENT 6	INSTRUMENT 6 cusip
7 R INSTRUMENT 7	INSTRUMENT 7 cusip
8 R INSTRUMENT_8	INSTRUMENT 8 cusip
9	

Open Calculator again and enter MYINDEX.allMembers(), which returns a set of the discrete members of the column "index_members.member_code". In this case, that means it returns a set of every Instrument that has ever been a member of MYINDEX.

Cal	Calculation Panel					
•						
4:	MYINDEX.allMembers()					
	ans4 = Collection of 8 objects of type GenericInstrument					

The scripts in this example create two data providers and import several Country and Currency models. The Currencies are resolved to models in the system using the built-in resolution properties, and Countries are resolved between the two data providers using a custom resolution property. The scripts also create two tables that provide data about the same models and then show how to import the data in each table as a single model rather than several (one from each table).

For more information about model resolution and resolution properties, see Model Overview.

1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data. Here is a sample of the database.

Provider One: "counti	ne"	Provider One: "countries_two"				Provider Two: "countries_three"		
name code numbe		number	name	рор	continent	area		
inter_provider_country	QN	2	intra_provider_country	North	North	256	name	number
intra_provider_country	QM	1	intra_provider_country	America		200	inter_provider_country	2

 Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

```
import com.palantir.finance.commons.data.model.*
```

```
Config {
   Database {
        driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
       username = "myUsername"
       password = "myPassword"
    }
}
Country() {
    // This adds the 'myResolutionNumber' property to the first tier of resolution
    // properties for the Country model type.
   resolutionProperties = [high: [['number']]]
    'countries_three' {
       modelSource = true
        id = [ columns: ['name'], tokens: ['name', 'symbol'] ]
       properties << 'number'
    }
}
```

3. Create a new file in /<your_palantir_server>/providers/prism and name it what you would like your second data provider to be called in Palantir. Add the following Carbon script to it, then change the login information in the Config block to match your database's login information.

```
import com.palantir.finance.commons.data.model.*
```

```
Config {
    Database {
        driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
        username = "myUsername"
```

```
password = "myPassword"
    }
}
Country() {
    'countries_one' {
       modelSource = true
        id = [ columns: ['name'], tokens: ['name', 'symbol'] ]
        properties << [code: 'countryCode'] << 'number'</pre>
    }
    'countries_two' {
       id = [columns: ['name']]
       properties << 'pop' << 'continent' << 'area'
    }
}
Currency() {
    'my_currencies' {
       modelSource = true
       id = [columns: ['currency_code'], tokens: ['name', 'symbol']]
        // The currencyCode property is the resolution property for the Currency
        // model type, so we have to specify it in order to resolve our USD to the
        // system's USD.
        properties << [my_prop: 'myProperty'] << [currency_code: 'currencyCode']</pre>
    }
}
```

4. (Re)start your Palantir instance's dispatch server and client. Open Explorer, set the starting universe to All models, add a Provider filter, and then select your data providers. Add a getType() column and sort by that. The ExplorerGroup panel at the bottom will show two Country models.

Note that inter_provider_country is provided by both of your data providers. The inter_provider_country models provided by the two data providers resolved to one because each model had the same value for our custom resolution property, number().

Also note that intra_provider_country is only displayed once, even though it appeared in two tables in the provider. Prism recognized that the two tables each provided a model with the same model ID (here, the value in the "name" column), so it imported those two models into Palantir as a single one. (For another example of this process, see Historical Membership.)

Explorer Group - 9 models Search:			
Name	Token	dataProviders	
intra_provider_country	QM	1 values: myExampleProvider2	
inter_provider_country	QN	2 values: myExampleProvider, myExampl	eProvider2
US Dollar	USD	3 values: common, myExampleProvider2,	, truths

If two models from the same provider have the same resolution properties, then neither will be exposed in Palantir (this is called blacklisting). For example, take two models that would resolve if they were imported from two different providers. If they were imported from the same provider instead, they would be blacklisted.

Open Calculator and enter USD.dataProviders(), then open the result in Table. Because our USD model resolved to the one in Palantir, the list of data providers will include your data provider. Then return to Calculator and enter USD.myProperty to return the custom data the scripts imported from the database.

Object Time Series - Strings

The scripts in this example creates a model and a metric that returns an object time series. The object time series contains

series of strings. An object time series is a time series whose values are objects (such as strings) rather than numbers.

- 1. Run this PostgreSQL script on your database. It will add the tables you need and fill them with example data.
- 2. Create a new file in /<your_palantir_sever>/providers/prism. The file's name will be the name of the data provider in Palantir.
- 3. Open the file and paste the following Carbon code (remember to change the login information in the Config block to match your database's login information).

```
import com.palantir.finance.commons.data.model.*
Config {
    Database {
        driver = "org.postgresql.Driver"
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
        username = "myUsername"
        password = "myPassword"
    }
}
GenericInstrument() {
    'hiring_events' {
        modelSource = true
        id = [columns: ['company'], tokens: ['name', 'symbol']]
        Timestamp {
            dateColumn = Column(name: 'start_date', dateFormat: FLEXDATE)
            metrics << [event: Metric(token: 'newHires', output:String.class)]</pre>
        }
    }
}
```

When you're done, open Calculator and enter MYCOMPANY.newHires(). Open the result in Table. You'll see a list of several recent hires as well as their start dates.

Date	Value	
1 2000-05-21	Hired Dopey	
2 2000-12-12	Hired Doc	
3 2004-09-23	Hired Sleepy	
4 2004-10-22	Hired Bashful	
5 2005-03-16	Hired Grumpy	
6 2005-06-11	Hired Happy	
7 2007-11-17	Hired Sneezy	
0		

Object Time Series - Models

This script creates two instruments STOCK1 and STOCK2. Each instrument has two metrics countrySeries() and stockSeries() that return object time series. The countrySeries() metric returns models of type Country and the stockSeries() metric returns models of type Stock.

- 1. Run this PostgreSQL script on your database to add three tables. The models that we return in the object time series must also be provided by the same provider. That's why our database includes tables for Country and Stock models.
- 2. Create a new file in /<your_palantir_sever>/providers/prism. The file's name will be the name of the data provider in Palantir.
- Open the file and paste the following Carbon code (remember to change the login information in the Config block to match your database's login information).

```
import com.palantir.finance.commons.data.model.*
Config {
    Database {
```

```
driver = 'org.postgresql.Driver'
        url = "jdbc:postgresql://myServerName:myPortNumber/myDatabaseName"
        username = "myUsername"
        password = "myPassword"
    }
}
Country {
    'countries' {
        modelSource = true
        id = [columns: ['country'], tokens: ['countryCode']]
        properties << [country: 'countryCode'] << [country: 'name']</pre>
    }
}
CommonStock {
    'stocks' {
        modelSource = true
        id = [columns: ['id', 'name'], tokens: ['symbol']]
        properties << [name: 'symbol'] << 'name'</pre>
        properties << [country: Property(token: 'countryTradingIn', output: Country.class)]</pre>
    }
}
GenericInstrument {
    'instruments' {
        modelSource = true
        id = [columns: ['symbol'], tokens: ['symbol']]
        properties << 'symbol' << [symbol: 'name']</pre>
        Timestamp {
            dateColumn = Column(name: 'ts', dateFormat: FULLDATE, frequency: DAYS)
            metrics << [country: Metric(token: 'countrySeries', output: Country.class)]</pre>
            metrics << [(['stock_id', 'stock']): Metric(token: 'stockSeries', output: CommonStock.class)]</pre>
        }
    }
}
```

Here's what is going on:

- The Country and CommonStock blocks create the models that will be returned in the object time series. We provide some
 properties for those models so that they will resolve to models from other data provider (if they exist). For example, most
 instances of Palantir include models that represent countries. The system will resolve two country models if they have the
 same countryCode property.
- The GenericInstrument block creates the two metrics:
 - The countrySeries() metric looks up the entries in the countries table based on 'country' column. In our table, some of the entries in 'instruments.country' are not valid because the countries 'QZ' and '' do not exist in 'countries.country'. These values are omitted from the object time series.
 - The stockSeries() metric looks up the entries in the stocks table by mapping the columns 'instruments.stock_id' and 'instruments.stock' to 'stocks.id' and 'stocks.name' respectively.

When you're done, open Calculator and try the following expressions:

- INSTRUMENT1.countrySeries()
- INSTRUMENT2.countrySeries()
- INSTRUMENT1.stockSeries()
- INSTRUMENT2.stockSeries()

Prism Overview Carbon Reference for Prism

Need Help? Email us at metropolis-support@palantir.com © 2013 Palantir Technologies · Terms of Use · Privacy and Security Statement

Metropolis DevZone > Data Integrator's Guide > Prism Overview > Carbon Reference for

Carbon Reference for Prism

Carbon is the scripting language for Prism configuration files. The language lets you specify how the Prism data provider maps the data from an external datasource to models and metrics in Palantir. Carbon is built on top of Groovy, which is a dynamic object-oriented language for the JVM and is byte-code compatible with Java.

If you are new to Prism, see Prism Overview.

This page covers the following topics:

- Intro to Groovy Syntax
- Importing Packages
- Variables
- Blocks
 - Configuration
 - Database
 - CSV Config
 - Table Config
 - Entitlement
 - Model Type
 - Table
 - Timestamp
- Built-in Functions
 - Property
 - Metric
 - Unit
 - BasicUnit
 - ColumnUnit
 - Parameter
 - Column
 - Documentation
 - Event

- Trigger
- Advanced Features
 - Triggers and Events
 - SQL Templates
 - Template Properties
 - Template Methods
- Recommended Code Editors
- See Also

Intro to Groovy Syntax

To build Prism providers, you need to know some Groovy syntax. This section covers some basics. We'll cover the rest as we go.

This page assumes that you understand programming concepts such as blocks, statements, maps, lists, and variables. For complete Groovy documentation, see Groovy documentation.

• **Strings** - Any value enclosed in either single or triple single quotes is a string. Strings often represent token or column names.

```
'This is also a string'
''' This is a
multi-line
string
'''
```

• Lists - A collection that holds a list of ordered objects of the same type.

[1, 2, 3] ['a', 'b', 'c']

• Maps - A collection that associates unique keys to values.

[1: 'a', 2: 'b', 3: 'c'] // Three entries. Number to String

If the key is a string, then omit the quotes.

[a: 1, b: 2, c: 3] // Three entries. String to Number.

If the key is an object, put parentheses around it.

[([1, 2, 3]): 'abc'] // One entry. List<Number> to String.

. Comments - Text that is ignored by Prism. Single line comments start with //

print 'hello' // This is a comment
/* This is a
 multi-line comment */

Importing Packages

Carbon scripts usually begin with **Import** statements. These statements let you refer to Model Types in your Carbon script without specifying their full package name. Most often, you will import com.palantir.finance.commons.data.model.* because it contains built-in model types such as CommonStock, Country, and Currency. But, you can import any package that is accessible by the server.

```
import com.palantir.finance.commons.data.model.*
// Now we can refer to CommonStock without package name
CommonStock{
    ...
}
```

Variables

You can define and use variables anywhere. However, we recommend assigning global variables after the Importing Packages statements for consistency. There are two types of variables in Carbon: **defined** variables and **built-in** variables.

• Defined Variables - These variable are create with the def keyword. Usually,

variables store constants or let you refactor and reuse pieces of code. Variables in a Block have local scope. Variables outside of any block have global scope.

```
// These are global variables
def myString = 'hey there'
def myMap = [key1: 'value1', key2: 'value2']
// These variables have local scope
CommonStock {
    def myUnit = BasicUnit('USD') / BasicUnit('share')
    def myClosure = { val -> val * val }
    def myMetric = Metric(token: 'testMetric')
}
```

 Built-in variables - These variables are created by the system, not the def keyword. The system is aware of these variables and customizes the data provider based on the values you assign. Most built-in variables have local scope, so we describe them later in each block. There is one built-in global variable:

Name	Туре	Required	Description
triggers	List <trigger></trigger>	N	A variable that holds a list of trigger objects for generating events. For more information, see Triggers and Events. There is also a triggers local variable in the Table block.

Blocks

A block is any section of code that has both a name and curly braces. For example, here is an empty Config block:

```
Config {
}
```

Many blocks have built-in variables and can hold other blocks. For example:

```
Config {
    Database {
        url = 'jdbc:postgresql://my-postgres-server:5432/mydatabase'
```

```
driver = 'org.postgresql.Driver'
}
```

In this example, the Database block is inside the Config block and it sets two mandatory built-in variables url and driver.

This section describes each of the blocks in Prism configuration files.

Configuration

The **configuration** block is always the first block in your Carbon file. It has the name **Config** and there is only one **Config** block per file. It contains settings that apply to the entire provider. This section describes the contents of the **Config** block.

Database

The **database** configuration block contains database connection info such as URL, username, and password. The system uses this information to connect to the database.

In version 3.14, the database configuration block is required. In versions 3.15 and later, omit this block if your provider exposes only CSV data.

The Database block has the following built-in variables:

Name	Туре	Required	Description
driver	String	Υ	One of the following supported JDBC driver namesp • org.postgresql.Driver • com.microsoft.sqlserver.jdbc.SQLServerDriver • oracle.jdbc.driver.OracleDriver • org.netezza.Driver
url	String	Y	The JDBC database connection URL string. Each dat vendor supports a different string format. For more information about formats for connection strings, se database's documentation. The strings typically look something like this:

			 jdbc:postgresql://server:5432/db jdbc:sqlserver://server:1433;DatabaseName= jdbc:oracle:thin:@server:1522:System_ID_of_ jdbc:netezza://server:5480/db
username	String	Y	Username string.
password	String	Y	Password string.
initialPoolSize	Integer	Ν	 The initial connection pool size. Palantir recommends an initial and minimum pc of 3-5 and a maximum of 50. You should adjus numbers according to your environment. For exa larger initial/minimum pool could result in bet performance because the system creates fewer connections at runtime, but it could also satura remote database with idle connections. For mol information on connection pool settings, see Or documentation on Optimizing Connection Pool.
minimumPoolSize	Integer	Ν	The minimum connection pool size.
maximumPoolSize	Integer	N	The maximum connection pool size.

Here is an example:

```
Config {
    Database {
        url = 'jdbc:postgresql://my-postgres-server:5432/mydatabase'
        driver = 'org.postgresql.Driver'
        username = 'us3rn@m3'
        password = 'p@ssw0rd'
        databaseInitialPoolSize = 3
        databaseMinimumPoolSize = 3
        databaseMaximumPoolSize = 48
    }
}
```

CSV Config

The **CSV Config** block contains information about your CSV files. This block is optional and is required only when your provider exposes CSV data. This block has the following built-in variables:

Name	Туре	Required	Description
rootDirectory	String	Y	A valid directory path that holds the CSV files.
nullValues	List <string></string>	Y	The strings that the system will treat as null values in the CSV file.

Here is an example:

```
Config {
   Csv {
      rootDirectory = ''
      nullValues = ['null', 'empty', 'n/a']
   }
}
```

Table Config

The **table** configuration block contains a single variable: the threshold value. This variable lets you adjust how the data provider fetches batched queries (a query that grabs multiple models) from the database. The block looks like this:

```
Config {
  Table {
    threshold = 0.3
  }
}
```

The threshold variable, which has a value from 0 to 1, represents a percentage value.

- If a batched query requests a higher percentage of the rows in the table than the threshold value, the provider fetches the entire database table and then filters out the unused entries.
- If the batched query requests a low percentage of the rows in the table than the threshold value, the provider creates a temporary table by joining the database

table with the items in the batch query, then fetching only the rows in the temporary table.

In Palantir 3.14, the default value is .2 (20%). In Palantir 3.15 and later, the default value .5 (50%).

Entitlement

By default, all Palantir users have access to the data in a Prism provider. The **entitlement** block lets you limit access to specific groups and/or users. This block has only one built-in variable:

Name	Туре	Required	Description
principals	Map <string,list<string>></string,list<string>	Y	A map of the authentication provider name to a list of group names. Palantir's default authentication provider is named 'builtin'.

Here is an example:

```
Config {
  Entitlement {
    principals = [ builtin: ['group1', 'group2']], ldap: ['group5'] ]
  }
}
```

This limits the access to builtin users in group1 and group2 and LDAP users in group5.

For information on managing authentication providers, users, and groups, see the AuthenticationPersister section.

The entitlement setting applies to the entire data provider; you cannot specify different settings for different pieces of data.

Model Type

Model type blocks specify the types of models to create. If your provider exposes more than one type of model, you need a model type block for each type. For example, the following provider exposes CommonStock and Country models:

```
import com.palantir.finance.commons.data.model.*
...
CommonStock {
    Table {
        ...
    }
}
Country {
    Table {
        ...
    }
}
```

Each Model type block contains one or more Table blocks. The table blocks supply the models with data, and usually, the system will create one model for each row in the table. We'll discuss this later in the Table section.

You can specify any type that implements the Model interface. Palantir has many built-in model types, but you can specify custom model types, as long as they have been plugged into the Palantir server. For more information on plugins, see Extensions and Plugins. If you are using a custom model, then you must specify the full class name as a string. For example:

```
'com.example.palantir.MyCustomModel' {
   Table {
        ...
    }
}
```

Model type blocks have two built-in variables:

Name	Туре	Required	Description
resolutionProperties	Map <string,list<list>></string,list<list>	Ν	This variable specifies a lists of hi added to the given model type's keywords are individually optiona in the map. For example: Country{

			<pre>resolutionProperties = low: [['lowCountry high: [['comboID1'] }</pre>
			The 'high' and 'low' keys have a l separate resolution tier within Pal properties will look like this:
			<pre>[[comboID1, comboID2], [hi [[countryCode]] [[countryCodeAlpha3]] [[lowCountryID]] < our</pre>
			If you don't specify resolutionPr model's built-in resolution behavic Overview.
			You must set this variable next to Instrument(resolutionTypes: }
resolutionTypes	List <class></class>	Ν	This variable lets you specify res model type with certainty. Instea system will attempt to resolve to models already instantiated by ot we're telling the system "This tak sure what type. They are either (resolve them to existing Commor then create the model as a Comn an interface and that both Comm is common to specify either an in then specify concrete subclasses

A **Table** block identifies a table in either a SQL database or a CSV file. For database tables, the block's name is the name of the table. For CSV tables, the name of the block is **File** and you must specify the **filename** variable. The table block defines the data in a model, and thus, must be inside a Model Type block. If the data for a model is spread over many tables, then specify multiple table blocks. For example:

CommonStock{

```
// This table holds property data
    File {
        filename = 'MyProperties.csv'
        modelSource = true
        id = [tokens: ['symbol'], columns:['model']]
        properties << ['name','description']</pre>
    }
    // This table holds timeseries data
    'TableWithMetricData'{
        id = [columns:['model']]
        Timestamp {
            dateColumn = Column(name: 'date', dateFormat: FLEXDATE)
            metrics << 'close' << 'volume'</pre>
        }
    }
}
```

Table blocks supports the following built-in variables:

Name	Туре	Required	Description
modelSource	Boolean	Ν	A boolean value that specifies whether this to called a model source if it supplies the initial model type. If multiple tables form the model must set modelSource = true. Each model bl acts as a model source. The default value is A Required Properties Palantir requires that every model have all model source blocks add the name()

			1
			properties variable or by adding it to th You should also add the properties that token properties (such as symbol() for in model visible in Palantir.
			Performance Tip Ideally, your table has exactly one row f that the model source table should not c is a database row for each tick in the tir columns are indexed by the database.
			Warning about NULL values Make sure your models do not contain n Otherwise, the system will ignore the m system. For example, if you have set id every row in the table must have values
			A map with two entries: columns and tokens, column names that identify the models in the then columns usually contains the primary ke id = [columns: ['Country', 'Symbol']
id	Map <string, List<string>></string></string, 	Y	This says that the models in our table are ide You can also specify an optional tokens entry where each string defines a property. The pr as the string and the property's value is the specified by columns. This is confusing so let
			<pre>id = [columns: ['Country', 'Symbol'] This creates two properties, id1 and id2 whi example, 'USA_MSFT' (the concatenation of)</pre>

properties	<pre>String List<string> Map<string column,="" or="" property="" string=""></string></string></pre>	N	This variable lets you add properties to the c are exposed as property metrics in the Palar operator << instead of the assignment operar properties in three ways: String - The string identifies a column i property whose token is the same as th properties << 'column1' // The p: List - The list contains a list of strings. the column and the property's token. Fc properties << ['column1', 'column Map - The map contains entries where value identifies a property. The column object. The property can either be a str properties << [column1: 'token1' (Column(name: 'columnName'))) Note on Map Syntax Groovy assumes that map keys are even though it is not surrounded by key is an object, it must be surrour (Column(name: 'columnName')). Note that the following three propertie: properties << ['myProperty1', 'm' properties << ['myProperty1' << '1 properties << ['myProperty1' : 'my] myProperty2 : Property(token
filter	String	N	This variable is valid for only database tables the selection of models to only a subset of rc

			optional and follows the syntax of a SQL WH // You must refer to columns as table
			filter = 'tableName.columnName IS NO
triggers	Trigger	N	A variable that holds a list of trigger objects information, see Triggers and Events.
sqlTemplate	String	N	This variable is valid for only database tables feature. For more information, see SQL Tem
aliases	Map <string,string></string,string>	N	This variable is used only when you also use information, see SQL Templates.

Timestamp

Timestamp blocks are always a child of a Table block and have the name Timestamp. These block supply data for properties and metrics that are associated with a date column. For example:

```
'TableWithMetricData'{
    id = [columns:['model']]
    Timestamp {
        dateColumn = Column(name: 'date', dateFormat: FLEXDATE)
        metrics << 'close' << 'volume'
    }
}</pre>
```

If your table has multiple 'date' columns, then specify multiple timestamp blocks under the same table block. The Timestamp block supports the following built-in variables:

Name	Туре	Required	Description
dateColumn	Column	Either dateColumn or dateColumns is required	A column object that identifies the date co dateColumn = Column(name: 'createDa
dateColumns	Map <string,column></string,column>	Either	This variable defines a date range with two

		dateColumn or dateColumns is required	<pre>The columns are used to create historical r dateColumns = [start: Column(name: end: Column(name: 'endDate', da The date range includes the start date, bu consistent with the rest of the platform.</pre>
properties	Map <string or<br="">Column, Property></string>	Ν	This variable lets you add properties that a membership to the data provider. The iten metrics in the Palantir platform. You must assignment operator. For example: properties << [propCol: Property(tc
metrics	<pre>String List<string> Map<string column,="" metric="" or="" string=""></string></string></pre>	Ν	<pre>This variable lets you add metrics to the d exposed as timeseries metrics in the Palan operator << instead of the assignment ope • String - The string identifies a colum whose token is the same as the colun metric << 'column1' // The meti • List - The list contains a list of strings column and the metric's token. For ex metrics << ['column1', 'column1' • Map - The map contains entries wher identifies a metric. The column can ei property can either be a string or a N metrics << [column1: 'token1',</pre>

	Note on Map Syntax Groovy assumes that map keys a though it is not surrounded by sin object, it must be surrounded by 'columnName')).
	Note that the following three metric s
	<pre>metrics << 'myMetricl' << 'myMe metrics << ['myMetricl', 'myMe metrics << [myMetricl : 'myMet] myMetric2 : Metric(token:'r</pre>

Built-in Functions

Carbon includes some built-in functions. These functions are used like constructors to create objects. The objects represent things like metrics, properties, and paramters that are used throughout Carbon scripts. Here is the syntax for calling functions:

```
Function( param1Name: value, param2Name: value )
```

Syntax Note for Unit Functions

Do not include parameter names when calling Unit functions. For example:

```
BasicUnit('USD')
ColumnUnit('CurrencyColumn')
```

This section describes each function in Carbon.

Property

The **Property** function creates a **Property** with specific attributes. Then, feed the object into the properties variable to expose it in the Palantir platform. For example:

```
CommonStock {
```

```
properties << [the_column: Property(token: 'myProp')]
}</pre>
```

The Property function accepts the following parameters:

Name	Туре	Required	Description
token	String	Y	The property's token (the name that appears in Palantir).
input	Class	N	The input type of the property. The class must implement Model.
output	Class	Ν	The output type of the property. You can specify Model types (such as Country, CommonStock, and Currency) as the output type, but any models returned by that property must be provided by the same provider. To return models from another provider, you must provide those models in your provider and then resolve them to the models in the other provider.
unit	Unit	Ν	The associated Unit object for this property.
metadata	Map <string, String></string, 	N	The metadata associated with this property. The map is a column to token mapping, and is only valid for tick-list properties.
multiValued	Boolean	N	A true or false value that indicates whether this property returns multiple values. The default is false.
timeSensitive	Boolean	N	A true or false value that indicates whether this property is a time-sensitive property. You must add Time-sensitive properties to the properties variable in the Timestamp block, not the Table block. The default is false.
batched	Boolean	N	A true or false value that indicates whether this property is associated with a date that

			is batched (i.e. indicating a batch of data within the database for a specific time period) and should pick either the value for a given date, or null if absent. The default is false.
isTickList	Boolean	N	A true or false value that indicates whether this property is a collection of ticks. The default value is false.
documentation	Documentation	N	The documentation for this property. Documentation appears in the metric flyout window.

Metric

The **Metric** function creates Metric with specific attributes. Then, feed the metric into the metrics variable to expose it in the Palantir platform. For example:

```
Timestamp {
    dateColumn = Column(name: 'the_date', dateFormat: FLEXDATE)
    metrics << [ the_column : Metric(token: 'myMetric') ]
}</pre>
```

The Metric function accepts the following parameters:

Name	Туре	Required	Description
token	String	Y	The metric token made visible in Palantir.
input	Class	Ν	Indicates the input type of the metric. The cla
output	Class	N	The type of objects in the time series. This va specify other object types like Strings. For an Time Series. You can specify Model types (suc Currency) as the output type, but any models To return models from another provider, you r provider and then resolve them to the models
parameters	List <parameter></parameter>	Ν	A list of Parameter objects for this metric. For

			def myParam = Parameter(name: 'paraml' def myMetric = Metric(token: 'metricV
metadata	Map <string, String></string, 	N	A map of metadata associated with this metric mapping.
unit	Unit	N	Associated unit object for this metric.
documentation	Documentation	N	The documentation for this metric. Documentation window.

Unit

The BasicUnit and ColumnUnit functions create a **Unit** object. A **Unit** represents a unit of measurement such as cubic meters or dollars per share.

BasicUnit

The **BasicUnit** function creates a Unit from a single string parameter such as kg, USD, or share. For more information on supported unit strings, see Acceptable Units of Measure. Call the function without specifying parameter name. For example: BasicUnit('JPY').

You can also combine units by dividing and multiplying them. The following example expresses the unit `JPY/share`:

```
def myUnit = BasicUnit('JPY') / BasicUnit('share')
metrics << [the_column: Metric(token: 'amountJPY', unit: myUnit )]</pre>
```

ColumnUnit

The **ColumnUnit** function creates a Unit (or multiple Units) by reading strings from a table column. You provide the column as a string parameter. Call the function without specifying parameter name. For example:

```
ColumnUnit('CurrencyColumn')
```

The column must contain valid unit string. For more information on supported unit strings, see Acceptable Units of Measure.

The BasicUnit and ColumnUnit functions both return Units. Thus, you can combine Units by dividing and multiplying them. For example:

```
def unit1 = (BasicUnit('USD') * ColumnUnit('unitColumn')) / BasicUnit('share'
```

Parameter

The **Parameter** function creates a Parameter that can be used in Metric objects. For example:

```
def myParam = Parameter(name: 'param1', column: 'metricParam1')
metric << [ the_column: Metric(token: 'metricWithParam', parameters: [myParam</pre>
```

The Parameter function accepts the following parameters:

Name	Туре	Required	Description
name	String	Y	The name of the metric parameter.
column	String	Υ	A valid database column to be used as a parameter.
type	Class	N	Specify this only if the parameter is a Model. Otherwise, the system automatically determines the type by reading the underlying column type.
propertyToken	String	N	Specifies the name (token) of the property that is used to identify the model. This variable is valid only if the parameter's type is a subclass of Model.
options	List <object></object>	N	Limits the values of this parameter to those in the list. The type of the objects in the list must match the parameter's type. If the parameter type is a model, then you cannot use options to specify a custom list. The

			autocomplete window will display all models of that type.
defaultOption	Object	N	Specifies a default value for the parameter. The type of this object must match the parameter's type.
optionsFromColumn	Boolean	Ν	If true, the parameter automatically populates the list of possible parameter values by reading the values in the database column.
description	String	N	The documentation for this parameter. Documentation appears in the metric flyout window.

Column

The **Column** function creates a Column with specific attributes. For example:

```
// Column in a property
def upperCaseColumn = Column(name: 'account_id',
    transform: { val -> if (val) {val.toUpperCase()} })
properties << [(upperCaseColumn): 'UpperCaseMetric']
// Column in a timestamp
Timestamp {
    dateColumn = Column(name: 'the_date_column', dateFormat: FULLDATE, freque
}</pre>
```

The Column function accepts the following parameters:

Name	Туре	Required	Description
			The name of the column. For CSV tables, column n in the first row.
name	String	Υ	Because support for mixed case column names vendor specific, you may encounter errors if you

			database contains mixed case column names. supports lowercase column names for all supp databases.
aggregation	String	N	A string that contains a valid SQL aggregation fundars, COUNT(), AVG(), MIN(), MAX(), SUM(), FIRST LAST().
dateFormat	DateTypeEnum	Ν	 When specifying a date column in a Timestamp bloc value indicates the format of the date. The format one of the following enumeration values: YEARMO - Specifies that the date type is a striformat YYYYMM. FULLDATE - Specifies that the date type is a ridatabase date-type. FLEXDATE - Specifies that the date type is a firstring value. Supported formats for this date yyyyMMdd dd-MM-yy yyyy-MM-dd HH:mm:ss dd-MMM-yy MM/dd/yyyy yyyy/MM/dd yyyyMM/dd yyyyMM/dd
frequency	FrequencyEnum	Ν	 When specifying a date column in a Timestamp bloc value indicates the frequency of the dates. You can one of the following enum values: NONE - Specifies that your data is intraday and recorded to the millisecond. Avoid time series that have ticks at consemilliseconds. The system considers a milling and the system considers and the system considers a milling and the system consystem considers a milling and

			 an instant, so the system could treat two consecutive ticks as a continuous time seq instead of two individual ticks. MINUTES - Specifies that your data is intraday recorded no more than once per minute and li minutely boundaries. DAYS - Specifies that your data is daily and is no more than once per day and lie on daily bo The system ignores times and time zones. The value is DAYS. When there are more than one value for a time point (say 2 values for the same day minute), MINUTES/DAYS disambiguates it picking the minimum value if the databasi are comparable. In NONE, all values are r
transform	Closure	Ν	A closure that is applied to the column value and retransformed value. For more information on closure Groovy documentation. For example, the following transforms a string to either the value zero or one: transform: { val -> val.contains("BPO") ?

Documentation

The **Documentation** function creates a **Documentation** object that adds descriptions, examples, and tags to your property or metric. For example:

```
// The documentation will appear in the metric flyout window
def desc = Documentation(
    name: "Balance of Bank Account",
    examples: ["_12345.balance(\"USD\")"],
    tags: ["bank"],
```

description: "Returns a time series that represents the balance of the sr

```
// The metric that uses the documentation.
def myMetric = Metric(
    token: "balance",
    parameters: [myParameter],
    documentation: desc)
```

The Documentation function accepts the following parameters:

Name	Туре	Required	Description
name	String	Υ	A brief summary of the metric, usually under 30 characters.
description	String	Ν	A description of the metric or property.
examples	List <string></string>	Ν	Examples of the metric in the Hedgehog language.
tags	List <string></string>	Ν	Tags to label the metric.

Here's what it looks like in the metric flyout window:

Balance of Bank Account			
Description Returns a time series that represents the bala	nce of the specified bank account.		
Examples _12345.balance("USD")			
Operates On:	Returns:		
GenericInstrument	TimeSeries		
Signature			
balance([String unitsParameter])			
Parameters (<i>default value</i>) String unitsParameter (<i>null</i>)			
Providers			
mark			
Tags			
bank			

Event

The **Event** function creates an **Event** object that can be used to set eventing triggers on the Prism provider. For example:

```
def eventsProperty = [ Event(entity: PROPERTY, scope: ALL) ]
def eventsMetric = [ Event(entity: METRIC, scope: ALL) ]
```

It accepts the following two parameters:

Name	Туре	Required	Description
entity	EventEntityEnum	γ	The entity that this event acts on. The valid enum values are: • MODEL • PROPERTY • METRIC.
scope	EventScopeEnum	Y	 The scope of the associated entity. The valid enum values are: TABLE - The scope of the entity is limited to the table where the event is used. ALL - The scope of the entity is global, i.e. it is not restricted to a single table.

For more information, see Triggers and Events.

Trigger

The **Trigger** function creates an eventing **Trigger** on the Prism provider. This triggers the associated events at the specified time, additionally querying the database if necessary. For example:

```
// refresh properties at 11:42pm daily
Trigger(name: 'propertyTrigger', cronTime: '0 42 23 * * ?', events: eventsPro
// refresh metrics at 11:45pm daily
Trigger(name: 'eventsTrigger', cronTime: '0 45 23 * * ?', events: eventsMetric)
```

It accepts the following parameters:

Name	Туре	Required	Description
name	String	Y	The name of the trigger.
cronTime	String	Υ	A valid cron trigger string. For more information on cron strings, see Cron documentation.
events	List <event></event>	Y	The list of events to be triggered.
column	Column	N	The column object that controls which specific events need to be scheduled in the Prism provider.

For more information, see Triggers and Events.

Advanced Features

This section covers some advanced features supported by Prism.

Triggers and Events

Triggers and events are used to update data (and cached data) in the Prism data provider. This is useful when your database is updated on a regular basis. To set a trigger, assign the trigger variable in either the global scope or Table block scope. The triggers variable contains a list of Trigger objects that indicate when to fire, which entities to fire on, and how to fetch those entities (for example, when an associated date or time value for a model changes, or a latest date value changes). Note that triggers don't support CSV data, so if your data provider must support triggers and events, you should use a different data format. Here is an example:

```
def eventsProperty = [ Event(entity: PROPERTY, scope: ALL) ]
def eventsMetric = [ Event(entity: METRIC, scope: ALL) ]
// Global trigger
triggers = [
    // refresh properties at 11:42pm daily
    Trigger(name: 'propertyTrigger', cronTime: '0 42 23 * * ?', events: event
    // refresh metrics at 11:45pm daily
    Trigger(name: 'eventsTrigger', cronTime: '0 45 23 * * ?', events: eventsN
]
```

Cron expressions must have 0 in the seconds field, or you will get an error.

SQL Templates

Prism lets you integrate many common database table designs without writing any SQL. For more complex designs, Prism exposes SQL underbelly through **SQL templates**. In this section, we will describe the templating mechanism.

Any table block that contains the variable sqlTemplate is interpreted as a user-defined SQL statement. sqlTemplate must contain a valid template string. An optional aliases variable holds a map of alias to table names used within the SQL template.

A SQL template is just a string that, when rendered, outputs a valid SQL statement. A template contains template items that can either be properties or methods. The rendering process takes a valid SQL template and substitutes each template item with its corresponding value. Here is an example ("' denotes a multi-line groovy string):

```
sqlTemplate = '''
SELECT a.idColumn,<<@csv fields>> FROM <<table>> AS
a JOIN <<tempTable>> AS t ON a.idColumn = t.idColumn ORDER BY column
```

Then, set the alias variable which contains a map of table aliases that are used in the SQL statement. This is necessary for properly validating properties/metrics and their associated data types. The key is used to indicate the alias name while the value is the actual table name. For example:

```
sqlTemplate = 'SELECT a.name FROM Instruments as a'
aliases << [a: 'Instruments']</pre>
```

Note that if the sql template uses temp tables, you don't not need to add them to the aliases map.

When creating a time-sensitive sqlTemplate block in Carbon, the sqlTemplate query must return ordered model IDs, hence the use of the ORDER BY clause in the previous example. All template items are surrounded by the << and >> operators. A property is identified by its name, for example, fields or tempTable. A method is indicated by the symbol @ followed by the method name, such as @csv. A method can only take in a single parameter, which can be another method (called method nesting) or a property.

Template Properties

SQL templates support the following properties:

Property	Туре	Description
table	String	The name of the table where the SQL template is defined
tempTable	String	The name of a temporary database table that contains the current set of model-ids being requested
ids	Set <string></string>	The current set of model-ids being requested
fields	Set <string></string>	A set of column names being requested. This includes property and/or metric columns along with any associated date/unit/parameter columns.
parameterFilters	String	A SQL conditional clause that can be used within `WHERE` to limit the set of rows being requested to only those that match any metric parameters. This is only useful if there are any metrics with parameters within the table block. For example: SELECT a.id, <<@csv fields>> FROM < <table>></table>
		AS a JOIN < <temptable>> as t ON a.id = t.id WHERE <<parameterfilters>></parameterfilters></temptable>

Template Methods

The following methods are also supported:

- @csv Operates on an input type Set<String>. Returns a comma-separated string of values.
- @quote Operates on an input type Set<String>. Returns another set of strings with each value enclosed within single quotes.

The template at the beginning of this page, when rendered on the models 123 and 456,

and properties name and symbol would generate the following SQL statement:

```
SELECT a.idColumn,a.name,a.symbol FROM
    exampleTable AS a JOIN exampleTable_t_56 AS T ON a.idColumn = t.idColumn
```

Note that the temporary table exampleTable_t_56 would contain the values 123 and 456 in its idColumn column.

Recommended Code Editors

You can use any text editor to create and modify Carbon files. However, because Carbon is built on top of Groovy, we strongly recommend using an editor that support syntax highlighting for Groovy, such as Sublime Text.

For a list of Groovy plugins for many major editors, see the official Groovy plugins page.

See Also Prism Overview Prism Examples

Need Help? Email us at metropolis-support@palantir.com© 2013 PalantirTechnologies · Terms of Use · Privacy and Security Statement