Pushkin Documentation

Release 1.0

Joshua Hartshorne

Jun 14, 2020

Contents:

CHAPTER 1

Starting a Pushkin Project

To start a new Pushkin website, you can install everything you need through the node project manager (npm). You will need to *install Node.js and npm <https://www.npmjs.com/get-npm>*.

FUBAR - UPDATE FOLLOWING

You will then need to set up the CLT. To do that, after you have cloned pushkin, move to pushkin's root directory and run:

` $ chmod +x pushkin_installCLT.sh $ ./pushkin_installCLT.sh `

This will also install the pushkin developer tools.

Note: These docs assume that the command 'pushkin' points to the CLT. If you choose not to do this, be aware that most of the docs will not work.

Pushkin relies on the following programs, which can easily be installed with Homebrew - if you're on a Mac - or another package manager: - node - npm - envsubst

Once these are installed, run `pushkin init` to automated installing packages and setting up the Pushkin environment.

Once you've got Pushkin downloaded and installed, see *Creating a Quiz* to make a quiz.

END FUBAR

CHAPTER 2

Updating Pushkin

FUBAR

CHAPTER 3

Pushkin Development

We don't recommend editing core Pushkin code locally, since this will make it difficult to update your distribution of Pushkin to take advantage of security patches, bug fixes, or new features. Instead, we recommend you fork the repository for the Pushkin tool in question. You can make a private npm package based on that repository. (If your changes are ones that others might want to make sure of as well, please submit a pull request!)

The main Pushkin repositories are: *pushkin-client* <https://github.com/pushkin-consortium/pushkin-client>‘ ——— A module that wraps around local-axios and provides simplified methods for making calls to a Pushkin API. Note that these built-in requests assume the API has default routes enabled. Documentation for the Pushkin Client is currently absent, however the experiment template available via the generate command from the CLI provides a showing of almost all the features in action.

3.1 *pushkin-api* <https://github.com/pushkin-consortium/pushkin-api>‘

Essentially a mini-server designed mainly with the use case of interfacing with Pushkin Client and Pushkin Worker. Once again, documentation is absent, but there is an example.

3.2 *pushkin-worker* <https://github.com/pushkin-consortium/pushkin-worker>‘

Installable via NPM. Adds a “pushkin” command to the path. No documentation available yet

3.3 *pushkin-cli* <https://github.com/pushkin-consortium/pushkin-cli>‘

Receives messages from RabbitMQ and runs whatever functionality it’s told to run, sending the result back through the queue it came from. Designed to be on the receiving end of a Pushkin API. Comes with built-in simple functions that most users will probably want, like “getAllStimuli”. Currently no explicit documentation, just an example.

3.4 *pushkin-jspsych* <https://github.com/pushkin-consortium/pushkin-jspsych>‘

The Pushkin fork of JSPsych makes a few small changes to the real JSPsych so that it can be bundled together as if it’s an NPM module. In order for it to be globally accessible to plugins as they expect, the import must be assigned to window.jsPsych. No documentation, present in example.

CHAPTER 4

Pushkin CLI

The Pushkin CLI comes packaged in the repo. Setup instructions can be found in *Starting a Pushkin Project*.

Variables relating to file structure and naming practices can be found in '.pushkin/pushkin_config_vars.sh'.

Pushkin has the following nested commands:

4.1 make

4.1.1 quiz

Creates a new quiz with all the required basic components. Pass the quiz name as an argument as follows: `pushkin make quiz [quiz name]`. Spaces and other special characters should be avoid. Note that the name seen by end users of the website can be changed to have special characters if needed by modifying the quizzes page in the front end. Generated quizzes are stored in `pushkin_user_quizzes`, 'quizzes/quizzes' by default.

4.1.2 compose

Creates the file specified by `pushkin_docker_compose_noDep_file`, replacing all docker variables set in `.env` as well as appending quiz compose files from the quizzes directory. Uses `pushkin_docker_compose_file` for the original compose file.

4.2 prep

Handles moving files and writing information to the various components of Pushkin infrastructure. This command allows for all quiz-related information to be consolidated in the quizzes folder.

It moves all quiz files from `pushkin_user_quizzes/[quiz name]` to their appropriate locations in the api, cron, front end, etc. and generates a quizzes.js (`pushkin_front_end_quizzes_list`) file in the front end.

4.3 compile

Compiles the front end using the command specified by `pushkin_front_end_compile_cmd`.

Moves the compiled files from `pushkin_front_end_dist` to `pushkin_server_html`.

4.4 migrate

Connects to the main database specified in `pushkin_env_file` and runs knex migrations from the db-worker's directory.

4.5 seed

Connects to the main database specified in `pushkin_env_file` and runs knex seeds from the db-worker's directory.

4.6 build

4.6.1 [core container]

Builds the docker container specified by [core container] where [core container] is one of "api", "cron", "dbworker", or "server".

4.6.2 core

Builds all the core docker containers.

4.6.3 quizzes

Builds each quiz's worker by looping through the `pushkin_user_quizzes` folder, each folder name being used as the quiz name. The tags given each quiz are templated as follows:

```
[image_prefix]/[quiz name][pushkin_user_quizzes_docker_suffix]:[image_tag]
```

where image_prefix and image_tag are specified in the docker '.env' file, `pushkin_user_quizzes_docker_suffix` is set in the pushkin config vars, and the quiz name based off the current folder in the quizzes directory.

4.6.4 all

Does both of the above steps.

4.7 sync

4.7.1 coreDockers

Pushes the api, cron, server, and db worker containers to docker hub.

4.7.2 quizDockers

Loops through the `pushkin_user_quizzes` folder and uses the same templating as in the `pushkin build quizzes` to push each image to docker hub.

4.7.3 website

Uses the AWS CLI to sync `pushkin_front_end_dist` with `s3_bucket_name`. Note that this means you must have installed and set up the AWS CLI.

4.7.4 all

Does all of the above steps.

4.8 start

A small convenience utility. Runs docker-compose up on `pushkin_docker_compose_noDep_file`.

4.9 init

Runs `npm install` in the api, front-end, and db-worker directories as specified by their variable names in `pushkin_config_vars.sh`.

CHAPTER 5

Set Up AWS

5.1 Requirements

These instructions assume you have:

- a working version of Pushkin (see *Starting a Pushkin Project*)
- access to AWS
- a Docker (Hub) account

5.2 Define security groups

You'll need two security groups: one for rancher and one for the databases.

The rancher EC2 security group should be open to all traffic on ports

Port	Protocol
80	TCP
443	TCP
8080	TCP
500	UDP
4500	UDP

This is for normal web access, the rancher management web UI, and the Rancher host infrastructure. The database security group should be open to all TCP traffic on ports 3306 (MySQL) and 5432 (Postgres).

Instructions on creating an EC2-security-group can be found on AWS. * Note: Link for linux instances *

<<<<<<< HEAD

>>>>>>> c3e5433b7e0799acba0b27a06014ae3af1704178 Instructions on creating a database-security-group can be found on AWS.

5.3 Get an EC2

The most straightforward way to do this is to use the official Rancher OS already on Amazon. Create it with the AMI from the list here appropriate to your region.

Instructions on creating EC2 instances can be found on AWS.

5.4 Get databases (RDS)

You'll need three databases. One's for rancher, one's for transactions, and one's for stored data, which we will refer to as Main DB. Launch all three with the database security group created previously. t2.medium is the recommended size for all of them.

- Rancher DB: MySQL
- Transaction DB: Postgres
- Main DB: Postgres

Instructions on creating RDS Instances can be found on AWS.

5.5 Prepare the transactions database

The transaction database serves as a running log of queries to the Main DB, recording all activity which passes through it.

Connect to the transactions database just created using any Postgres client and run the following code to make a transactions table:

```
create table transactions (
   id  SERIAL PRIMARY KEY,
   query TEXT not null,
   bindings TEXT
)
```

5.6 Get an S3 Bucket

Next, you'll need an S3 bucket for file backups, data storage, and retrieval. Create a new S3 bucket with open permissions for all traffic.

Instructions on creating S3 buckets can be found on AWS.

5.7 Get CloudFront

Now, you will need an AWS CloudFront distribution. CloudFront connects to an S3 bucket, and serves to distribute the contents of that bucket, which could include pickled objects, data-sets, and other resources, to any web application which possesses a CloudFront URL.

This URL is provided to Pushkin in the .env file located in the root folder.

Instructions on creating CloudFront distributions can be found on AWS.

5.8 Get AWS CLI Tools

Follow Amazon's instructions for installing the AWS CLI here. Alternatively, you could use Homebrew if you're on a Mac.

5.9 Set up IAM Users and Roles

You will need some way of securely controlling access to AWS services and resources. This can be done by setting up IAM roles and users, which allows other developers and contributers to access your resources without needing to share access keys or passwords.

More information on IAM users can be found **here_**

CHAPTER 6

Set Up Rancher

If you haven't *Set Up AWS* yet, do that first.

6.1 Login to Rancher

SSH into the Rancher EC2 instance and start the docker container for Rancher. Replace the capitalized parts of the following command with the information for the rancher database created earlier.

```
sudo docker run -d --restart=unless-stopped --name=rancher -p 8080:8080␣
↪rancher/server --db-host DB_URL --db-port 3306 --db-user DB_USER --db-pass␣
↪DB_PASSWORD --db-name DB_NAME
```

You should now be able to connect to Rancher's web interface by going to the EC2 URL at port 8080.

6.2 Add a Password

Go to Admin > Access Control and set up an access control type of your choice.

6.3 Add a host

Go to Infrastructure > Hosts > Add Host. Use the public IP of the current Rancher EC2 instance for the public IP of the host and run the command given in the SSH connection already open.

6.4 Set Variables

The ".env" file in the root directory of Pushkin is used to house the configuration of a myriad of docker settings. Open it in a plain text editor and enter in the corresponding information for each line.

6.5 Create a new stack

Go to Stacks > New Stack in the Rancher web UI and upload the docker-compose file generated for you (called "docker-compose.production.noEnvDependency.yml" by default). If this doesn't exist, make sure you've made a quiz (*Creating a Quiz*) and done the initial deployment steps (*Preparing for Deployment*).

CHAPTER 7

Creating a Quiz

Creating quizzes on Pushkin is straightforward. Start in the root of the Pushkin directory and follow the below steps.

Make and setup the basics of a quiz by running the following commands.

pushkin make [quiz name]

There is now a new folder in quizzes/quizzes called [quiz name]. Inside the db_migrations folder are files detailing the default database structure. The db_seeds folder contains initial data to insert into the database. Modify either of these to best fit your quiz, or leave them alone to be dealt with later. Once you're satisfied with them, run `pushkin prep` to update the files and then `pushkin migrate` and `pushkin seed` to connect to the main database with connection information in `pushkin_env_file`.

Next, run the following commands to build the docker containers and create a finalized version of the docker compose file that contains your new quiz. See *Pushkin CLI* to learn more about these commands.

pushkin build all pushkin make compose

CHAPTER 8

Foundational Quiz Components

8.1 Front-end Page

Under the folder 'quiz_page' is housed the React component(s) of a Pushkin quiz. When a user visits the quiz page of the website and clicks a link to a quiz, the default export from index.js is loaded and served on a blank canvas to give over full control of the page.

8.2 Database Preparation Process

Before a quiz can be run, and data recorded and stored, the database must contain the appropriate tables, and be seeded with an array of stimuli to present to quiz-takers. This task is handled by files contained in the root db-worker folder.

8.2.1 Database Migrations

Under the migrations folder, you will find several timestamped files for each Pushkin quiz. Each migration file serves to define and create the columns of a database table, by specifying the names and valid data types of each column. Each database table deals with a different aspect of quiz data. Take a look at each of them to see how the database is laid out. Here's the code for the default users table. You might like to add columns to keep track of more demographics

```
exports.up = function(knex) {
  return knex.schema.createTable('bloodmagic_users', table => {
    table.increments('id').primary();
    table.string('auth0_id');
    table.timestamp('created_at');
    table.timestamp('updated_at');
    table.date('dob');
    table.string('native_language');
  });
};
```

8.2.2 Database Seeds

The next step is to seed the database, which means adding the initial data. In most cases, this will simply mean adding stimuli to the stimulus table. You should edit this file to add your own stimuli to the database. The quiz's front-end page expects to receive stimuli in JSON format that can be added to its JsPsych timeline.

Note: Make sure to edit the quiz files inside your quiz folder under the root 'quizzes' directory. Otherwise it may get overwritten by `pushkin prep`, which expects quiz files to be inside the quizzes directory.

8.3 Cron Scripts

Cron is a language-agnostic (meaning that code execution is not limited to a subset of programming languages) service for running programming scripts on a scheduled, periodic basis. In the context of Pushkin, cron occupies its own docker container, with its own dependencies, and is composed of two main components:

- Crontab

  This is a configuration file which schedules shell commands for execution. Each line of the crontab specifies a single job, and that job's schedule.

  These sample tasks are executing python scripts, and saving their output (If any) to .txt files.

```
# Execute every 5 minutes.

  5 * * * * root echo "test" >> /scripts/log.txt

# Execute at time 00:00 (midnight) every day.

  0 0 * * * root /usr/bin/python2.7  /scripts/test.py >> /scripts/log.txt

# Execute at 10:00 on the first day of every month.

  0 10 1 1 * root /usr/bin/python2.7  /scripts/secondTest.py >> /scripts/out.txt

# Execute every minute on Monday only.

  1 * * * 1 root /usr/bin/python2.7  /scripts/testBoto.py >> /scripts/out2.txt
```

  This system of scheduling is powerful and easy-to-use.

Note: Asterisks are wildcard symbols that default to the max value of that field. For more information and to easily generate your own crontab from a friendly interface, see the *crontab guru <https://crontab.guru>*.

```
# ┌──────────── minute (0 - 59)
# │ ┌──────────── hour (0 - 23)
# │ │ ┌──────────── day of month (1 - 31)
# │ │ │ ┌──────────── month (1 - 12)
# │ │ │ │ ┌──────────── day of week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                       7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * *  command to execute
```

- Scripts

  The jobs themselves can be written in any programming language, and can perform any required task on schedule. Cron's Dockerfile is set by default to load everything in the scripts directory.

  These scripts may be useful for periodically organizing or analyzing data. Docker provides this container access to your database via an enviroment variable called 'DATABASE_URL', which encodes the username and password as set in the '.env' file as well.

- DockerFile

  This file is responsible for establishing the environment of your docker container, installing necessary dependencies and packages by running shell commands. For example, the following three commands install curl, then pip, then boto3 for python.

  - RUN apt-get install curl -y
  - RUN curl –silent –show-error –retry 5 https://bootstrap.pypa.io/get-pip.py | python
  - RUN pip install boto3

8.4 API Controller

The API controller establishes communication endpoints between the front-end, represented by the quiz and user interface, and the back-end, which consists of the database and associated workers. Each endpoint serves as an interface which allows the frontend to make HTTP requests to the core of Pushkin.

A POST endpoint from a quiz controller:

```
{ path: '/health', method: 'health', queue: db_read_queue },
```

And the corresponding switch and method in the quiz worker's handlers:

```
case 'health':
// no data fields to require
return this.health();
.......
health() {
  return Promise.resolve({ message: 'healthy' });
}
```

Should you need to add your own custom endpoints, simply add a path in your quiz's API controller, a corresponding case in the worker's handler, and whatever functions that method will need.

CHAPTER 9

Frontend Quiz Components

9.1 jsPsych Plugins

Each trial requires a jsPsych plugin to provide the assets and functions necessary to run a certain experimental paradigm. The quiz template page contains the following require statements, which point towards the jsPsyc folder contained in front-end/src/quizzes/libraries :

```
# This makes jsPsych core code available. It is needed for all plugins.
require('../libraries/jsPsych/jspsych.js');

# This makes code associated with a single plugin available. It is needed for a trial␣
↪of a specific type.
require('../libraries/jsPsych/plugins/jspsych-instructions.js');
```

Plugins are used as templates for single trials. They offer a range of question types, from multiple choice and likert scales to custom variants which can be created and added as required.

For more information, please refer to jsPsych's official documentation.

9.2 Creating Trials

Trials are defined and located in `quiz_files/jsPsychTimeline.js`. This file exports the trials as a single array, referred to as a timeline. The timeline is fed to a jsPsych init function, which then proceeds to execute the trials in order.

Below is a sample trial. It is helpful to reference the source code of the plugin which you wish to use, in the jsPsych folder, in order to understand the requirements of the trial. In general, each can be described as an object with defined parameters, typically provided with strings of HTML/CSS for formatting, and arrays of strings, images, and audiofiles to serve as stimuli/answer options for that trial.

Note: This is not correct. I don't have time to write it all out, but, as I've (Jacob) shown Mariela, stimuli for trials are stored in the database and then retrieved through calls to the API. Axios posts to /startExperiment, then /getStimuli

and sends the retrieved info to the "buildTimeline" function in jspTimeline.js. I don't know why Han wrote this this way.

```
# A relatively simple trial which serves only to display instructions.
const intro = {
    type: 'instructions',
    pages: [" <p align='left'> A sample paragraph written in HTML! </p>"],
    show_clickable_nav: true,
    button_label_next:'Continue'
};

# A more complicated trial designed to provide feedback and social-media-sharing␣
↪options.
var const testingTrial = {
    type: 'display-prediction',
    prompt1: "Nuestras tres mejores conjecturas para su lengua materna:",
    prompt2: "Nuestras tres mejores conjecturas para su dialecto español:",
    prediction1: ['Guess1', 'Guess2', 'Guess3'],
    prediction2: ['Guess11', 'Guess22', 'Guess33'],
    buttonText: 'Terminar',
    quizURL: 'http://www.gameswithwords.org/WhichEnglish/',
    subjectLine: 'Mapeando la gramática española por el mundo entero',
    teaserPart1: "Ayudé a GamesWithWords.org a entrenar su algoritmo a adivinar qué␣
↪español hablo. Adivinó que mi lengua materna es ",
    teaserPart2: " y que ",
    teaserPart3: " es mi dialecto. Cual español hablas?",
    socialSharing: false,
    encourageDemographics: 'Por favor continúa para contestar algunas preguntas y␣
↪para ayudarnos a entrenar nuestro algoritmo!',
    mailButtonImg: `${baseUrl}/quizzes/email.png`,
    fbButtonImg: `${baseUrl}/quizzes/fb.png`,
    twitterButtonImg: `${baseUrl}/quizzes/twitter.png`,
    weiboButtonImg: `${baseUrl}/quizzes/weibo.png`,

    # Take note of this recordData function. This tells jsPsych what to do with data
    # recently collected from a trial.

    on_finish: data => {
                        recordData(data);
                        }
    }
}
```

To create a new trial, simply declare it in `jsPsychTimeline.js`, and then choose where to insert it into the timeline.

Each trial, or type of trial, also requires a methods for making an Axios call to the API controller.

9.3 Saving Data

Data saving is carried out automatically by Axios calls within `jsPsychTimeline.js`. Advanced users should attempt to edit the endpoints to provide functionality other than basic database reading and writing.

```
# This function serves to place an Axios call to a pre-defined endpoint, passing␣
↪along the user's ID number
# and the data from the trial as a JSON string, to be processed by the methods␣
↪attached to that endpoint in
# the quiz API controller.

const recordData = function(data){
    return axios
            .post('/stimulusResponse', {
                user_id: self.props.user.profile.id,
                data_string: data,
            })
            .then(function(res) {
                self.props.dispatchTempResponse({
                    user_id: self.props.user.profile.id,
                    data_string: data,
                });
            })
}

# This illustrates a timeline containing only a single trial, with a data recording␣
↪function attached at the bottom.

timeline = [
    {
    type:"survey-multi-choice",
    required:[true],
    preamble: ['Click on the word that comes closest in meaning to the word in all␣
↪CAPS:'],
    questions:[stimuli[i].stimulus],
    options:[stimuli[i].options.split(", ")],
    correct:[stimuli[i].correct],
    horizontal:false,
    force_correct:false,
    on_finish: function(data){
        recordData(data)
    }
]
```

CHAPTER 10

Preparing for Deployment

Once you have installed Pushkin (*Starting a Pushkin Project*), made a quiz running locally (*Creating a Quiz*), and setup AWS (*Set Up AWS*) you're ready to deploy to the web.

Run:

```
pushkin prep
pushkin compile
pushkin build all
pushkin sync all
```

to prep, build, and push all files online. This may take a few minutes. Once it's done, run `pushkin make compose` to create a Docker compose file without any environment variables that includes your custom quiz workers, suitable for use with Rancher. The file generated will be called 'docker-compose.production.noEnvDependency.yml' by default, or whatever you've set `pushkin_docker_compose_noDep_file` to.

10.1 Deploying Pushkin Locally

Once you've gone through the process of creating and compiling a quiz, you can proceed to locally deploy Pushkin, in order to freely develop and test quizzes and other site features.

First, run `pushkin start` to start Pushkin locally.

Now, we need to seed the database for any quizzes which we wish to test or develop. Run `docker ps` and find the name of the DB worker container, which is 'db-worker_1' by default.

```
CONTAINER ID        IMAGE                COMMAND                  CREATED          STATUS␣
↪         PORTS               NAMES

be3c744c9a81        pushkin_db-worker    "bash start.debug.sh"  23 hours ago     Up 23␣
↪hours                          pushkin_db-worker_1
```

Now, copy that container name and run `docker exec -it [name] bash` to get a shell inside the DB worker Docker container. Type `npm run migrations` to run the migrations, then `npm run seed` to populate the database with your seeds. Type `exit` to return to your normal shell.

> **Caution:** If you have multiple quizzes and have added data to a table in the database that a seed file corresponds to, be aware that seeding removes all data in the table before beginning. Make sure to first delete the files inside the seeds folder of the db-worker container that you don't want to run.

We're almost done. The last step is to find your server container. Run `docker ps` again, and this time look for the server container.

```
CONTAINER ID        IMAGE               COMMAND                  CREATED        STATUS
↪    PORTS                     NAMES

1744091332f1        pushkin_server    "nginx -g 'daemon ..." 24 hours ago  Up 7␣
↪seconds    0.0.0.0:54328->80/tcp   pushkin_server_1
```

This time, we're most interested in the port. Select the port which points to 80/tcp. In this example, this is port `54328`. You may now open a browser, and enter `http://localhost:[Port Number]` to access the local deployment of Pushkin.

All done! The quiz has been made, its assigned database tables properly seeded, and the Pushkin platform is up and running.

For more information on the key parts of a Pushkin quiz, see *Foundational Quiz Components*.

CHAPTER 11

Maintenance

Learn how to maintain and troubleshoot Pushkin.

11.1 Logs

Logs for docker containers can be viewed by running:

```
docker logs [container]
```

where [container] is the name of the docker container you'd like to see logs of. Running containers and their names can be listed with:

```
docker ps
```

CHAPTER 12

Mariela's Notes

ReST and MD files are mutually exclusive on the current RTD setup and porting takes a little time. See the original here:

https://github.com/l3atbc-datadog/pushkin/blob/master/docs/source/old/tutorial.md

CHAPTER 13

Indices and tables

- genindex
- search