Abstract

JBrowse 2 is a platform for genomics data visualization. JBrowse 2 is packaged as a web-app that can be setup on your server or as an executable that can be run on your desktop. It also has re-usable components that can be embedded in your web apps. This document includes a user manual with screenshots of how to use the app, a configuration guide for administrators, and developer documentation for programmers making JBrowse 2 plugins or core code modifications.

Contents

JBrowse desktop quick start .......................... 7
  Installing JBrowse desktop .................................. 7
  Installing on Windows ..................................... 7
  Installing on MacOS .................................... 7
  Installing on Linux ..................................... 7
  JBrowse Desktop start screen .................................. 7
    Special features on the start screen ..................... 8
    Next steps ........................................ 8

JBrowse web setup using the CLI ....................... 10
  Prerequisites ........................................ 10
  Installing the JBrowse CLI .................................. 10
    Using jbrowse create to download JBrowse 2 ..................... 10
    Checking the download .................................. 10
  Running JBrowse 2 .................................... 11
  Adding tracks ........................................ 12
    Adding a genome assembly in FASTA format ............... 12
    Adding a BAM or CRAM track .................................. 13
    Adding a VCF track .................................... 14
    Adding a BigWig/BigBed track .................................. 15
    Adding a GFF3 file with GFF3Tabix ....................... 15
    Adding a synteny track from a PAF file ..................... 15
  Indexing feature names for searching ..................... 16
  Conclusion ........................................ 16
  Miscellaneous tips ...................................... 16
    Upgrade JBrowse Web to the latest version ................. 16
    Upgrade @jbrowse/cli to the latest ...................... 17
    Output to a custom named config file, and output to subfolders ...... 17

JBrowse web setup using the GUI ....................... 19
  Pre-requisites ........................................ 19
  Starting JBrowse 2 admin server ......................... 19
  Adding a genome assembly .................................. 19
    Editing a genome assembly .................................. 21
  Adding a track ........................................ 21
Overview

Overview

Overview

Introduction and overview .............................................. 30
Products and plugins .................................................. 30
Example plugins .......................................................... 31

Configuration model basics ............................................ 33
Configuration slot types .................................................. 33
Example config with multiple slot types ................................. 33
Accessing config values .................................................. 34
Using config callbacks ................................................... 34
Example of a config callback ............................................. 34
Configuration internals ................................................... 35
Creating a custom adapter ................................................. 36
  What is an adapter ....................................................... 36
  What types of adapters are there .................................... 36
  Skeleton of a feature adapter ....................................... 37
  Example feature adapter .............................................. 37
  What is needed from a feature adapter .............................. 38
Creating a display type .................................................. 39
Creating a custom renderer .............................................. 40
  What is a renderer ...................................................... 40
  How to create a new renderer ....................................... 40
  What are the props passed to the renderer ......................... 41
  Adding custom props to the renderer ............................... 41
  Rendering SVG ........................................................ 42
  Overriding the renderer’s getFeatures method ..................... 43
Creating a custom track type ............................................ 43
Examples of track types .................................................. 44

Creating a custom view type .......................................... 46

Extension points .......................................................... 48
  Core-extendPluggableElement ....................................... 48
  Core-guessAdapterForLocation ..................................... 48
  Core-guessTrackTypeForLocation ................................... 48
  Core-extendSession ................................................... 48
  Core-customizeAbout .................................................. 49
  TrackSelector-multiTrackMenuItems ................................ 49
  LaunchView-LinearGenomeView ..................................... 49
  LaunchView-CircularView ............................................ 49
  LaunchView-SvInspectorView ....................................... 49
  LaunchView-SpreadsheetView ....................................... 49
  LaunchView-DotplotView ............................................. 50
  Core-replaceAbout .................................................... 50
  Core-extraAboutPanel .................................................. 51
  Core-customizeAbout ................................................ 51
  Core-replaceWidget .................................................... 51
Core-extraFeaturePanel .................................................................................. 52
Extension point footnote .................................................................................. 52

Pluggable elements ........................................................................................... 54
  Pluggable elements .......................................................................................... 54
    View types ..................................................................................................... 54
    Adapters ....................................................................................................... 55
    Track types ................................................................................................... 55
    Displays ........................................................................................................ 55
    Renderers ....................................................................................................... 55
    Widgets ......................................................................................................... 56
    RPC methods ............................................................................................... 56
    Add track workflows .................................................................................... 56
    Extension points ........................................................................................... 57
  Next steps ......................................................................................................... 57

URL query parameter API .................................................................................. 60
  Simple API for linear genome view ................................................................ 60
    ?config= ...................................................................................................... 60
    &assembly= ............................................................................................... 60
    &loc= .......................................................................................................... 60
    &tracks= ..................................................................................................... 60
  More URL parameters ...................................................................................... 61
    &sessionTracks= ......................................................................................... 61
    &session= ................................................................................................... 61

API ................................................................................................................... 66
  MenuItems objects .......................................................................................... 66
  rootModel Menu API ...................................................................................... 68

Installing and using plugins ............................................................................ 72
  Prerequisites .................................................................................................... 72
  Installing plugins with one click ..................................................................... 72
    GDC plugin .................................................................................................. 72
    ICGC plugin ................................................................................................. 79
  Configuring plugins ......................................................................................... 81
    UCSC plugin ................................................................................................ 81
    Biothings plugin .......................................................................................... 82
    CIVIC plugin ............................................................................................... 82
  Conclusion ......................................................................................................... 85

Developing a simple JBrowse plugin ................................................................. 87
  Prerequisites .................................................................................................... 87
  Use git to clone the plugin template ............................................................... 87
  Initialize the project ........................................................................................ 87
  Setup JBrowse 2 .............................................................................................. 87
  Add new files, stubs, and install dependencies .............................................. 89
    A widget’s index.tsx ..................................................................................... 89
    A widget’s ReactComponent ....................................................................... 91
  Install the plugin to JBrowse at runtime ....................................................... 92
  Setup the configuration for proper testing .................................................... 94
  Testing it out .................................................................................................... 96
    Run JBrowse with your new plugin and manually test ............................... 96
  Writing a simple integration test with cypress ............................................ 97
    Add a cypress test ....................................................................................... 97
    Running cypress ......................................................................................... 98
  Next steps ..................................................................................................... 100
Developing with JBrowse web and desktop
Prerequisites .............................................. 105
Setup JBrowse 2 using the latest developer build .... 105
To run JBrowse web ................................... 105
To run JBrowse desktop ............................... 106
Running JBrowse desktop with a plugin in development . 106
Next Steps ............................................... 109

Writing a no-build plugin
Prerequisites .............................................. 111
What is the difference between a no-build plugin and a regular plugin? .............................. 111
Writing a no-build plugin .............................. 111
Adding a callback function which you can use in your config ................................................ 111
Adding a global menu item ............................ 112
Importing with jbrequire ................................ 113
Conclusion and next steps ............................. 114

Writing a desktop specific plugin
Prerequisites .............................................. 116
Why might I need a desktop-specific plugin ................................. 116
Writing a desktop-specific plugin .................... 116
Setup ..................................................... 116
Running CLI operations through JBrowse desktop .................................................. 118
Testing the plugin ....................................... 124
Running the plugin on JBrowse desktop .......... 124
Distributing the plugin ................................ 125
Next steps ................................................ 125
Core takeaways ......................................... 125

Embedded components
JBrowse React Linear Genome View .................. 127
JBrowse React Circular Genome View .............. 127
JBrowse Jupyter ......................................... 128
JBrowseR ............................................... 128

Command line tools
Installation .............................................. 130
Commands ............................................... 130
jbrowse add-assembly SEQUENCE ................ 130
jbrowse add-connection CONNECTIONURLORPATH .................................................. 133
jbrowse add-track TRACK ............................ 134
jbrowse add-track-json TRACK .................... 135
jbrowse admin-server ................................ 136
jbrowse create LOCALPATH ......................... 136
jbrowse help [COMMAND] ............................. 137
jbrowse remove-track TRACK ....................... 137
jbrowse set-default-session ......................... 138
jbrowse text-index .................................... 139
jbrowse upgrade [LOCALPATH] ...................... 140
Debugging ................................................. 141
FAQ

Developers ................................................................. 143
  How can I start the JBrowse 2 app as a developer ...................... 143
General ................................................................. 143
  What is special about JBrowse 2 .................................... 143
  What are new features in JBrowse 2 ................................ 143
Setup ................................................................. 143
  What web server do I need to run JBrowse 2 .......................... 143
  BAM files do not work on my server ................................ 144
  How can I setup JBrowse 2 on my web server .......................... 144
  How do I install or update the @jbrowse/cli tool ...................... 144
  How can I make a header on a jbrowse-web instance .................... 144
  How do I update my instance of jbrowse-web .......................... 144
  How can I setup JBrowse 2 without the CLI tools ...................... 144
  How do I load a track into JBrowse 2 ................................ 145
  How do I customize the color of the features displayed on my track 145
  My jexl is too complicated, how can I simplify it? .................. 145
Adding color callbacks in the GUI .................................... 145
Adding color callbacks via the command line .......................... 145
Curiosities ............................................................ 146
  Why do all the tracks need an assembly specified .................... 146
  How are the menus structured in the app ........................... 146
  Why do some of my reads not display soft clipping .................... 146
  Do you have any tips for learning React and mobx-state-tree ....... 146
  What technologies does JBrowse 2 use ................................ 146
  Should I configure gzip on my web server ........................... 146
  How does JBrowse know when to display the “Zoom in to see more features” message 146
Text searching ......................................................... 148
  Why I am running out of disk space while trix is running ............ 148
  How does the jbrowse text-index trix format work .................... 148
URL params .............................................................. 148
  Why can’t I copy and paste my URL bar to share it with another user 148
  How does the session sharing work with shortened URLs work in JBrowse Web 148
Troubleshooting ......................................................... 149
  What should I do if the Share system isn’t working? ................ 149
**JBrowse desktop quick start**

In this guide, we’ll get the JBrowse desktop application running on your computer.

### Installing JBrowse desktop

#### Installing on Windows

Click here to download the latest Windows installer executable.

Double-click the downloaded installer and it will install and open JBrowse. You can now open JBrowse like any other program.

#### Installing on MacOS

Click here to download the latest MacOS release artifact.

When the .dmg file is downloaded, double click, and drag JBrowse 2 into ‘applications’.

You can now open JBrowse 2 like any other application on your Mac.

#### Installing on Linux

Click here to download the latest Linux AppImage release.

Start it in one of two ways:

**In the terminal**

Using the AppImage file on linux, all that is needed is to make the file executable which you can do in a terminal

```bash
# Make the AppImage file executable, only need to do this once
chmod a+x jbrowse-desktop-*-linux.AppImage
# Run!
./jbrowse-desktop-*-linux.AppImage
```

**In your file explorer**

This may vary slightly depending on your distribution but should basically follow these steps:

1. Right-click on the AppImage file and select “Properties”
2. Go to the “Permissions” tab
3. Check “Allow executing file as program” and close

You can now double-click the AppImage file to launch JBrowse.

### JBrowse Desktop start screen

After you have installed and started JBrowse Desktop you will see a start screen like this:
On the left hand panel, “Launch new session” can launch a new session using either your own custom genome (which you can load using an indexed FASTA or a twobit file via open sequence file) or a pre-loaded genome via the “Quickstart list”.

On the right hand panel is the “Recently opened sessions”. This includes sessions that you have explicitly saved, and sessions that were autosaved (i.e. ones that you didn’t explicitly use “Save as” on). You can re-open your sessions by clicking on the session name.

Special features on the start screen

Converting a saved session into a quickstart entry  If you study a rare species, you might find it useful to customize your quickstart panel. You can convert a session in the “Recently opened sessions” into an entry that appears in the quickstart list.

To do this: Click a checkbox next to a session in the “Recently opened sessions” table, and then hit the icon next to the trash can icon. This is helpful if you want to make your own custom organism a template for quickstarts in the future.

Selecting multiple entries from the quickstart panel Users can also hit the checkbox for multiple species in the quickstart list, and then the sessions are combined which can be helpful for comparative genomics.

Next steps

Check out the user guide for more info on how to use JBrowse Desktop. This covers all the features that are available with screenshots and instructions.
JBrowse web setup using the CLI

Prerequisites

- Node.js 12+. Note: we recommend not using \texttt{apt} to install Node.js, it often installs old versions. Good alternatives include NodeSource or NVM.
- Samtools installed e.g. \texttt{sudo apt install samtools} or \texttt{brew install samtools}, used for creating FASTA index and BAM/CRAM processing for creating tabix GFF
- tabix installed e.g. \texttt{sudo apt install tabix} and \texttt{brew install htslib}, used for creating tabix indexes for BED/VCF/GFF files
- (optional) \texttt{genometools} installed e.g. \texttt{sudo apt install genometools} or \texttt{brew install brewsci/bio/genometools}, used for sorting GFF3. can use awk instead of \texttt{genometools} instead

Installing the JBrowse CLI

The JBrowse CLI can help perform many tasks to help you manage JBrowse 2, such as:

- create a new instance of JBrowse 2 automatically
- update an existing instance of JBrowse 2 with the latest released version
- configure your JBrowse 2 instance

To globally install the JBrowse CLI, run

\texttt{npm install -g @jbrowse/cli}

After running this command you can then test the installation with

\texttt{jbrowse --version}

which will output the current version of the JBrowse CLI.

If you can’t or don’t want to globally install the JBrowse CLI, you can also use the \texttt{npx} command, which is included with Node.js, to run JBrowse CLI without installing it. Simply replace \texttt{jbrowse} with \texttt{npx @jbrowse/cli} in any command, e.g.

\texttt{npx @jbrowse/cli --version}

Using \texttt{jbrowse create} to download JBrowse 2

In the directory where you would like to download JBrowse 2, run

\texttt{jbrowse create jbrowse2}

This fetches the latest version of jbrowse-web and unzips it to a folder named “jbrowse2” from github (https://github.com/GMOD/jbrowse-components/releases), you could run this step manually if you wanted to instead.

Checking the download

The directory where you downloaded JBrowse should look something like this:

\texttt{jbrowse2/}
\|-- asset-manifest.json
\|-- favicon.ico
\|-- index.html
\|-- manifest.json
\|-- robots.txt
\|-- static/
\|-- test_data/
\|-- version.txt
Running JBrowse 2

JBrowse 2 requires a web server to run. It won’t work if you try to directly open the index.html in your web browser.

Oftentimes, you may put the folder on a web server in the static html folder e.g. /var/www/html/jbrowse2/ once in place, you can then visit http://yours server/jbrowse2

You could also use a simple server to check that JBrowse 2 has been downloaded properly. Run

```bash
cd jbrowse2/
npx serve .
# or
npx serve -S . # if you want to refer to symlinked data later on
```

which will start a web server in our JBrowse 2 directory.

Navigate to the location specified in the CLI’s output (likely http://localhost:3000).

Your page should look something like this:

Configuration not found. You may have arrived here if you requested a config that does not exist or you have not set up your JBrowse yet.

Sample JBrowse configs:

- Volvox sample data
- Human basic
- Human sample data
- Tomato SVs
- Breakpoint
- Grape/Peach dotplot
- Grape/Peach synteny
- Yeast synteny
- Many contigs
- Honeybee
- Wormbase
- WGBS methylation

No config.json found. If you want to learn how to complete your setup, visit our quick start guide.

Figure 2: JBrowse 2 screen showing no configuration found

Click on the sample config to see JBrowse 2 running with a demo configuration. It should look like this:
Congratulations! You’re running JBrowse 2.

Adding tracks

Now that JBrowse 2 is set up, you can configure it with your own genomes and tracks.

Adding a genome assembly in FASTA format

The first step to creating a jbrowse config is to load a genome assembly. This is normally in FASTA format, and we will start by creating a “FASTA index” with `samtools`:

```
samtools faidx genome.fa
jbrowse add-assembly genome.fa --load copy --out /var/www/html/jbrowse/
```

This will output a configuration snippet to a file named `/var/www/html/jbrowse/config.json` if it does not already exist, or append a new assembly to that config file if it does exist. It will also copy genome.fa and genome.fa.fai to the `/var/www/html/jbrowse/` folder because we used --load copy. If you wanted to symlink instead, can use --load symlink

JBrowse 2 also supports other assembly file formats, including bgzip-compressed indexed FASTA, and 2bit files.
Adding a BAM or CRAM track

For this example we will use a BAM file to add an alignments track. As with assemblies, you can add a track using local files or remote locations of your files.

```bash
samtools index file.bam
jbrowse add-track file.bam --load copy --out /var/www/html/jbrowse
samtools index file.cram
jbrowse add-track file.cram --load copy --out /var/www/html/jbrowse
```

This will add a track configuration entry to /var/www/html/jbrowse/config.json and copy the files into the folder as well. If you use –load symlink, it can symlink the files instead. To see more options adding the track, such as specifying a name, run jbrowse add-track --help.

If you have JBrowse 2 running as described in the JBrowse web quickstart, you can refresh the page and add a linear genome view of the volvox assembly. Then open track selector, and you will see the alignments track.
Adding a VCF track

Adding a variant track is similar to adding an alignments track. For this example, we will use a VCF file for the track. JBrowse 2 expects VCFs to be compressed with `bgzip` and `tabix` indexed.

To add the track, run

```
bgzip file.vcf
tabix file.vcf.gz
jbrowse add-track file.vcf.gz --load copy --out /var/www/html/jbrowse
```

Note if you get errors about your VCF file not being sorted when using tabix, you can use bcftools to sort your VCF.

```
bcftools sort file.vcf > file.sorted.vcf
bgzip file.sorted.vcf
tabix file.sorted.vcf.gz
```

You can also bgzip and index with the `bcftools` tool.

```
bcftools view volvox.vcf --output-type z > volvox.vcf.gz
rm volvox.vcf
bcftools index --tbi volvox.vcf.gz
```

For more info about `bgzip`, `tabix`, and `bcftools`, see https://www.htslib.org/.

Figure 5: JBrowse 2 linear genome view with alignments track
Adding a BigWig/BigBed track

Probably one of the most simple track types to load is a BigWig/BigBed file since it does not have any external index file, it is just a single file.

```
jbrowse add-track file.bw --load copy --out /var/www/html/jbrowse
```

Adding a GFF3 file with GFF3Tabix

To load a GFF3 file, we can sort and index it with tabix, make sure you have GenomeTools (to install can use `sudo apt install genometools`).

```
gt gff3 -sortlines -tidy -retainids yourfile.gff > yourfile.sorted.gff
ggzip yourfile.sorted.gff
atabix yourfile.sorted.gff.gz
jbrowse add-track yourfile.sorted.gff.gz --load copy
```

As an alternative to `gt gff3 -sortlines`, use `awk` and GNU `sort`, as follows:

```
awk '$1 !~/^#/ { print $0; next } { print $0 | sort -t "\t" -k1,1 -k4,4n"}' file.gff > file.sorted.gff
ggzip file.sorted.gff
atabix file.sorted.gff.gz
```

The `awk` command is inspired by the method in the tabix documentation, but avoids subshells and properly sets the tab delimiter for GNU sort in case there are spaces in the GFF.

Adding a synteny track from a PAF file

Loading synteny data makes use of all the previous functions we’ve used so far in this guide.

Here, we make use of the grape and peach genome assemblies, but replace with your own data if applicable.

Use minimap2 to create a PAF file from FASTA files:
## Use minimap2 to create a PAF from your assemblies

```
minimap2 grape.fa peach.fa > peach_vs_grape.paf
```

## add each assembly to jbrowse config
## the -n flag names the assemblies explicitly

```
jbrowse add-assembly grape.fa --load copy -n grape --out /var/www/html/jbrowse
j/browse add-assemble peach.fa --load copy -n peach --out /var/www/html/jbrowse
```

Next, we’ll load the synteny “track” from the PAF file.

**Order matters here for the --assemblyNames parameter:**

If minimap2 is run as minimap2 grape.fa peach.fa, then you need to load as --assemblyNames peach,grape.

The order is reversed between the minimap2 and jbrowse tools.

```
jbrowse add-track peach_vs_grape.paf --assemblyNames peach,grape --load copy --out /var/www/html/jbrowse
```

### Indexing feature names for searching

The final step of loading your JBrowse instance may include adding a “search index” so that you can search by genes or other features by their name or ID.

To do this we can use the jbrowse text-index command:

```
jbrowse text-index --out /var/www/html/jbrowse
```

This will index relevant track types e.g. any track with Gff3TabixAdapter (gene names and IDs) or VcfTabixAdapter (e.g. variant IDs). The command will print out a progress bar for each track that it is indexing.

This will also update your config.json so that after it completes, you can type a gene name into the “search box” in the linear genome view or other views and quickly navigate to genes by gene name.

See the text-index command docs for more info. Also see the FAQ entries for text searching.

### Conclusion

Now that you have JBrowse configured with an assembly and a couple of tracks, you can start customizing it further. Check out the rest of the docs for more information, especially the JBrowse CLI docs for more details on some of the steps shown here.

### Miscellaneous tips

You can use --subDir to organize your data directory:

```
mkdir my_bams
## Copies .bam and .bai files to my_bams folder
j/browse add-track myfile.bam --subDir my_bams --load copy --out /var/www/html/jbrowse
```

### Upgrade JBrowse Web to the latest version

You can upgrade your JBrowse release to the latest version with:

```
# run this command in an existing jbrowse 2 installation
j/browse upgrade
```

The above command downloads the latest jbrowse-web from github and unzips it into the current directory
Upgrade @jbrowse/cli to the latest

To upgrade the CLI tools, you can re-run the install command

```
npm install -g @jbrowse/cli
```

Output to a custom named config file, and output to subfolders

You can use filenames other than config.json, and put configs in subfolders of your jbrowse 2 installation too

```
jbrowse add-assembly mygenome.fa --out /path/to/my/jbrowse2/subfolder/alt_config.json --load copy
```

This would then be accessible at e.g. http://localhost/jbrowse2/?config=subfolder/alt_config.json
**JBrowse web setup using the GUI**

In order to display your data, JBrowse 2 needs to know about the reference genome for your organism of interest and needs to have tracks created that reference your data sources. This guide will show you how to set those up using JBrowse 2's graphical configuration editing.

You can also do this configuration with JBrowse CLI. See that guide here.

**Pre-requisites**

This tutorial requires having the following software installed

- JBrowse CLI
- JBrowse 2 web application

**Starting JBrowse 2 admin server**

The JBrowse CLI contains a tool called *admin-server*. This will act as a web server for JBrowse 2 and will write any changes made in JBrowse 2 to a config file.

:::warning

Note The *admin-server* is meant to be used temporarily for configuration, **not in production**. :::

The *admin-server* launches an instance of JBrowse 2 in “admin mode”, which then lets you:

- Add and edit assemblies with the “Assembly manager”
- Add tracks and edit tracks
- Add and edit connections

All of these changes will be written by the server to the JBrowse config file (usually *config.json*) located in the JBrowse instance. This is something that can only be done while the *admin-server* is running, which again, is only meant to be temporary!

To start the *admin-server*, navigate into your JBrowse 2 directory and run:

```
## Start the admin-server
jbrowse admin-server
```

This will then generate a link that you can visit in your web browser:

![JBrowse CLI admin-server output](image)

Figure 7: JBrowse CLI admin-server output, which provides a link that can be used.

**Adding a genome assembly**

A key first step in configuring a genome browser is adding an assembly to view. In order to do this, use the navigation bar to open up the Assembly Manager (Admin > Open Assembly Manager).

This opens up a table which can be used to create, edit, and delete assemblies in your application:
As an example, let’s add the hg38 human reference genome to our JBrowse 2 application.

Press the “Add New Assembly” button, and enter the necessary information in the form:

- **name:** hg38
- **type:** BgzipFastaAdapter
- **fasta:** https://jbrowse.org/genomes/GRCh38/fasta/hg38.prefix.fa.gz
- **fasta index:** https://jbrowse.org/genomes/GRCh38/fasta/hg38.prefix.fa.gz.fai
- **gzi:** https://jbrowse.org/genomes/GRCh38/fasta/hg38.prefix.fa.gz.gzi

Click on “Create New Assembly”. Great, we’ve added an assembly! We can see that we have successfully added the hg38 assembly:
The assembly can be edited or deleted, but for now we will return to the application.

**Editing a genome assembly**

After you’ve added a genome assembly, you can use the pencil icon button in the Assembly manager to edit that assembly. You can also delete assemblies from the assembly manager.

**Adding a track**

To add a new track or connection, you can use the menu bar in the app to open the form for adding a track:

File > Open Track:
Alternatively, you can use the action button (circular “+”) inside the track selector to access the “Add track” form:

In the “Add track” form, you can provide a URL or select a local file to load.

The following file formats are supported in core JBrowse 2:

- CRAM
- BAM
- htseq
- VCF (Tabix-indexed)
- GFF3 (Tabix-indexed)
- BED (Tabix-indexed)
- BigBed
- BigWig
- JBrowse 1 nested containment lists (NCLists)
- plain text VCF, BED, CSV, TSV, BEDPE, STAR-fusion output (tabular formats)
- PAF (synteny/dotplot)
- Indexed FASTA/BGZip indexed FASTA
- 2bit
- .hic (Hi-C contact matrix visualization)

Additional data formats can be supported via plugins; checkout the plugin store.

For tabix files, TBI or CSI indexes are allowed. CSI or BAI is allowed for BAM. Only CRAI is allowed for CRAM. The index will be inferred for BAI or TBI files as e.g. filename+.bai. If it is different from this, make sure to specify the index file explicitly.

**Editing a track**

First, open a Linear Genome View using the navigation bar (*File > Add > Linear Genome View*), and click on the “Select Tracks” button.

The configuration settings are accessible by clicking on the ellipses by each track.

![Figure 13: The configuration editor, which will persist settings to the config file if editing using the admin-server.](image)

Open the configuration editor for the track by clicking on the “Settings” button shown above. You can use the configuration editor to live-edit any configurable value for a given track.

**Setting a default session**

It is also possible to use the graphical admin server to set the default session of your JBrowse 2 instance. This is the session that will appear when JBrowse 2 is first visited. To do so, open the form to set the default session (*Admin > Set default session*):
Figure 14: The ‘Set default session’ will persist your current session into the config file so any subsequent visitors to the app will see this session.

You can use the form to clear your default session, select the currently open session, or any of your previously saved sessions.

**Additional resources**

There are a number of additional features for configuring JBrowse 2. Make sure to refer to the config guide for more topics.

**Conclusion**

This guide showed how to launch the admin-server in the JBrowse CLI to perform graphical configuration of your application. Specifically, we looked at how to access and use the assembly manager, as well as how to access the configuration editor for tracks. Importantly, all tracks have different configuration options available in the configuration editor.

Make sure to take a look at any tracks you add to JBrowse 2 that you might want to further configure!
Overview

This user guide is a comprehensive compilation of all general UI navigation, tools/features available for some of the different track types, and other useful features for using JBrowse.
Overview

The following guide provides comprehensive information regarding the anatomy and usage of the `config.json` file that is critical for running a JBrowse 2 session.

Also see our quick-start tutorials for configuring assemblies and tracks CLI or the GUI.
Overview

This guide will introduce the JBrowse 2 ecosystem from the developer’s point of view. We’ll examine the core concepts of how code is packaged and structured, and then go over how to create new plugins and pluggable elements.

Introduction and overview

Let’s get a high-level view of the JBrowse 2 ecosystem.

Products and plugins

The JBrowse 2 ecosystem has two main types of top-level artifacts that are published on their own: products and plugins.

A “product” is an application of some kind that is published on its own (a web app, an electron app, a CLI app, etc). jbrowse-web, jbrowse-desktop, and jbrowse-cli are products.

A “plugin” is a package of functionality that is designed to “plug in” to a product at runtime to add functionality. These can be written and published by anyone, not just the JBrowse core team. Not all of the products use plugins, but most of them do.

Figure 15: Architecture diagram of JBrowse 2, showing how plugins encapsulate views (e.g. LinearGenomeView, DotplotView etc.), tracks (AlignmentsTrack, VariantTrack, etc.), adapters (BamAdapter, VcfTabixAdapter, etc.) and other logic like mobx state tree autoruns that add logic to other parts of the app (e.g. adding context menus)
Figure 16: This figure summarizes the general architecture of our state model and React component tree.

Example plugins

You can follow this guide for developing plugins, but you might also want to refer to working versions of plugins on the web now.

This repo contains a template for creating new plugins https://github.com/GMOD/jbrowse-plugin-template.

Here are some examples of working plugins:

- jbrowse-plugin-ucsc-api probably the simplest plugin example, it demonstrates accessing data from UCSC REST API
- jbrowse-plugin-gwas a custom plugin to display manhattan plot GWAS data
- jbrowse-plugin-biothings-api demonstrates accessing data from mygene.info, part of the “biothings API” family
- jbrowse-plugin-msaview - demonstrates creating a custom view type that doesn’t use any conventional tracks
- jbrowse-plugin-gdc demonstrates accessing GDC cancer data GraphQL API, plus a custom drawer and track type for coloring variants by impact score
- jbrowse-plugin-systeminformation demonstrates using desktop specific functionality, accessing system node libraries. This desktop specific functionality should use the CJS bundle type (electron doesn’t support ESM yet)

You can use these to see how plugins are generally structured, and can use the pluggable elements in them as templates for your own pluggable elements.
Configuration model basics

This guide will introduce some critical concepts to understanding the configuration model. The configuration model is used to structure the features and data available in a given JBrowse session, and each new pluggable element you create will need its own configuration schema.

Configuration slot types

Our configuration system is “typed” to facilitate graphical editing of the configuration. Each configuration has a “schema” that lists what “configuration slots” it has. Each configuration slot has a name, description, a type, and a value.

Here is a mostly comprehensive list of config types:

- **stringEnum** - allows assigning one of a limited set of entries, becomes a dropdown box in the GUI
- **color** - allows selecting a color, becomes a color picker in the GUI
- **number** - allows entering any numeric value
- **string** - allows entering any string
- **integer** - allows entering a integer value
- **boolean**
- **frozen** - an arbitrary JSON can be specified in this config slot, becomes textarea in the GUI
- **fileLocation** - refers to a URL, local file path on desktop, or file blob object in the browser
- **text** - allows entering a string, becomes textarea in the GUI
- **stringArray** - allows entering a list of strings, becomes a “todolist” style editor in the GUI where you can add or delete things
- **stringArrayMap** - allows entering a list of key-value entries

Let’s examine the `PileupRenderer` configuration as an example.

Example config with multiple slot types

This `PileupRenderer` config contains an example of several different slot types:

```typescript
// plugins/alignments/src/PileupRenderer/configSchema.ts

import { types } from 'mobx-state-tree'

export default ConfigurationSchema('PileupRenderer', {
  color: {
    type: 'color',
    description: 'the color of each feature in a pileup alignment',
    defaultValue: jexl.get(feature, 'strand') == -1 ? '#8F8F8F' : '#EC8B8B',
    contextVariable: ['feature'],
  },
  displayMode: {
    type: 'stringEnum',
    model: types.enumeration('displayMode', ['normal', 'compact', 'collapse']),
    description: 'Alternative display modes',
    defaultValue: 'normal',
  },
  minSubfeatureWidth: {
    type: 'number',
    description: 'the minimum width in px for a pileup mismatch feature. use for increasing mismatch marker widths when zoomed out to e.g. 1px or 0.5px',
    defaultValue: 0,
  },
  maxHeight: {
    type: 'integer',
  }
})
```
description: 'the maximum height to be used in a pileup rendering',
    defaultValue: 600,
},
})

Accessing config values

So instead of accessing config.displayMode, we say,

```
readConfObject(config, 'displayMode')
```

You might also see in the code like this:

```
getConf(track, 'maxHeight')
```

Which would be equivalent to calling,

```
readConfObject(track.configuration, 'maxHeight')
```

Using config callbacks

Config callbacks allow you to have a dynamic color based on some function logic you provide. All config slots can actually become config callback. The arguments that are given to the callback are listed by the 'contextVariable' but must be provided by the calling code (the code reading the config slot). To pass arguments to the a callback we say

```
readConfObject(config, 'color', { feature })
```

That implies the color configuration callback will be passed a feature, so the config callback can be a complex function determining the color to use based on various feature attributes.

Example of a config callback

We use Jexl to express callbacks. See https://github.com/TomFrost/Jexl for more details.

There are also more examples and information in our config guide.

If you had a variant track in your config, and wanted to make a custom config callback for color, it might look like this:

```
{
    "type": "VariantTrack",
    "trackId": "variant_colors",
    "name": "volvox filtered vcf (green snp, purple indel)",
    "category": ["VCF"],
    "assemblyNames": ["volvox"],
    "adapter": {
        "type": "VcfTabixAdapter",
        "vcfGzLocation": {
            "uri": "volvox.filtered.vcf.gz",
            "locationType": "UriLocation"
        },
        "index": {
            "location": {
                "uri": "volvox.filtered.vcf.gz.tbi",
                "locationType": "UriLocation"
            }
        }
    },
}
```
"displays": [
    {
      "type": "LinearVariantDisplay",
      "displayId": "volvox_filtered_vcf_color-LinearVariantDisplay",
      "renderer": {
        "type": "SvgFeatureRenderer",
        "color1": "jexl: get(feature,'type')=='SNV'? 'green': 'purple'" # here we call our jexl function
      }
    }
]
}

This draws all SNV (single nucleotide variants) as green, and other types as purple (insertion, deletion, other structural variant).

You can also write your own jexl function and call it in the same way in the configuration.

:::info Note
It can be extremely useful to utilize a custom jexl function in the default configuration for a pluggable element type, as you can observe in one of the examples above, the default value of the color slot of the renderer is a jexl function. If you configure your plugin with a custom jexl function, you can use that function as a default value in your various pluggable elements.

:::

### Configuration internals

A configuration is a type of mobx-state-tree model, in which leaf nodes are ConfigSlot types, and other nodes are ConfigurationSchema types.

```plaintext
Schema
/   \ 
Slot Schema Slot
   / \ 
Slot Slot
```

Configurations are all descendants of a single root configuration, which is root.configuration.

Configuration types should always be created by the ConfigurationSchema factory, e.g.:

```javascript
import { ConfigurationSchema } from '@jbrowse/core/utils/configuration'
const ThingStateModel = types.model('MyThingsState', { 
  foo: 42,
  configuration: ConfigurationSchema('MyThing', { 
    backgroundColor: {
      defaultValue: 'white',
      type: 'string',
    },
  }),
});
```

An example of a config schema with a sub-config schema is the BamAdapter, with the index sub-config schema:

```javascript
ConfigurationSchema(
  'BamAdapter',
  {
    bamLocation: {
      type: 'fileLocation',
    },
  })
```
Reading the sub-config schema is as follows

```javascript
const indexType = readConfObject(config, ['index', 'indexType'])
```

Alternatively can use

```javascript
const indexConf = readConfObject(config, ['index'])
indexConf.indexType
```

However, this may miss default values from the slot, the `readConfObject` has special logic to fill in the default value.

Creating a custom adapter

What is an adapter

An adapter is essentially a class that fetches and parses your data and returns it in a format JBrowse understands.

For example, if you have some data source that contains genes, and you want to display those genes using JBrowse’s existing gene displays, you can write a custom adapter to do so. If you want to do a custom display of your data, though, you’ll probably need to create a custom display and/or renderer along with your adapter.

What types of adapters are there

- **Feature adapter** - This is the most common type of adapter. Essentially, it takes a request for a region (a chromosome, starting position, and ending position) and returns the features (e.g. genes, reads, variants, etc.) that are in that region. Examples of this in JBrowse include adapters for BAM and VCF file formats.
- **Regions adapter** - This type of adapter is used to define what regions are in an assembly. It returns a list of chromosomes/contigs/scaffolds and their sizes. An example of this in JBrowse is an adapter for a chrome.sizes file.
- **Sequence adapter** - This is basically a combination of a regions adapter and a feature adapter. It can give the list of regions in an assembly, and can also return the sequence of a queried region. Examples of this in JBrowse include adapters for FASTA and .2bit file formats.
- **RefName alias adapter** - This type of adapter is used to return data about aliases for reference sequence names, for example to define that “chr1” is an alias for “1”. An example of this in JBrowse is an adapter for (alias files)[http://software.broadinstitute.org/software/igv/LoadData/#aliasfile]
- **Text search adapter** - This type of adapter is used to search through text search indexes. Returns list of search results. An example of this in JBrowse is the trix text search adapter.
:::info Note When using the refName alias adapter, it’s important that the first column match what is seen in your FASTA file. :::

**Skeleton of a feature adapter**

A basic feature adapter might look like this (with implementation omitted for simplicity):

```javascript
class MyAdapter extends BaseFeatureDataAdapter {
    constructor(config) {
        // config
    }
    async getRefNames() {
        // return refNames used in your adapter, used for refName renaming
    }
    getFeatures(region, opts) {
        // region: {
        //     refName: string, e.g. chr1
        //     start: number, 0-based half open start coord
        //     end: number, 0-based half open end coord
        //     assemblyName: string, assembly name
        //     originalRefName: string the name of the refName from the fasta file, e.g. 1 instead of chr1
        // }
        // opts: {
        //     signal?: AbortSignal
        //     ...rest: all the renderProps() object from the display type
        // }
    }
    freeResources(region) {
        // can be empty
    }
}
```

So to make a feature adapter, you implement the `getRefNames` function (optional), the `getFeatures` function (returns an rxjs observable stream of features, discussed below) and `freeResources` (optional).

**Example feature adapter**

To take this a little slow, let’s look at each function individually.

This is a more complete description of the class interface that you can implement:

```javascript
import { BaseFeatureDataAdapter } from '@jbrowse/core/data_adapters/BaseAdapter'
import SimpleFeature from '@jbrowse/core/util/simpleFeature'
import { readConfObject } from '@jbrowse/core/configuration'
import { ObservableCreate } from '@jbrowse/core/util/rxjs'

class MyAdapter extends BaseFeatureDataAdapter {
    // your constructor gets a config object that you can read with readConfObject
    // if you use "subadapters" then you can initialize those with getSubAdapter
    constructor(config, getSubAdapter) {
        const fileLocation = readConfObject(config, 'fileLocation')
        const subadapter = readConfObject(config, 'sequenceAdapter')
        const sequenceAdapter = getSubAdapter(subadapter)
    }
}
```
What is needed from a feature adapter

getRefNames  Returns the refNames that are contained in the file. This is used for “refname renaming” and is optional, but highly useful in scenarios like human chromosomes which have, for example, chr1 vs 1.

Returning the refNames used by a given file or resource allows JBrowse to automatically smooth these small naming disparities over. See reference renaming.
**getFeatures**  
A function that returns features from the file given a genomic range query e.g.,

`getFeatures(region, options)`

The region parameter contains:

```typescript
interface Region {
  refName: string
  start: number
  end: number
  originalRefName: string
  assemblyName: string
}
```

The `refName`, `start`, `end` specify a simple genomic range. The `assemblyName` is used to query a specific assembly if your adapter responds to multiple assemblies, e.g. for a synteny data file or a REST API that queries a backend with multiple assemblies.

The `originalRefName` are also passed, where `originalRefName` is the queried refname before ref renaming e.g. in BamAdapter, if the BAM file uses chr1, and your reference genome file uses 1, then originalRefName will be 1 and refName will be chr1.

The options parameter to `getFeatures` can contain any number of things:

```typescript
interface Options {
  bpPerPx: number
  signal: AbortSignal
  statusCallback: Function
  headers: Record<string, string>
}
```

- `bpPerPx` - number: resolution of the genome browser when the features were fetched
- `signal` - can be used to abort a fetch request when it is no longer needed, from AbortController
- `statusCallback` - not implemented yet but in the future may allow you to report the status of your loading operations
- `headers` - set of HTTP headers as a JSON object
- anything from the `renderProps` of the display model type gets passed to the `getFeatures` opts

We return an rxjs `Observable` from `getFeatures`. This is similar to a JBrowse 1 `getFeatures` call, where we pass each feature to a `featureCallback`, tell it when we are done with `finishCallback`, and send errors to `errorCallback`, except we do all those things with the `Observable`

Here is a “conversion” of JBrowse-1-style `getFeatures` callbacks to JBrowse 2 observable calls

- `featureCallback(new SimpleFeature(...))` -> `observer.next(new SimpleFeature(...))`
- `finishCallback()` -> `observer.complete()`
- `errorCallback(error)` -> `observer.error(error)`

**freeResources**  
This is uncommonly used, so most adapters make this an empty function

Most adapters in fact use an LRU cache to make resources go away over time instead of manually cleaning up resources

### Creating a display type

Display types tell JBrowse how to “display” a given track in a particular view. A track might “display” itself completely different depending on whether it is in a dotplot or in a linear genome view. The “display” types may not actually do the drawing of the track data: that is often done by the renderer. The display will call the renderer though.

Here are some reasons you might want a custom display type:
• Drawing custom things over the rendered content (e.g. drawing the Y-scale bar in the wiggle track)
• Implementing custom track menu items (e.g. Show soft clipping in the alignments track)
• Adding custom widgets (e.g. custom `VariantFeatureWidget` in variant track)
• You want to bundle your renderer and adapter as a specific thing that is automatically initialized rather than the `BasicTrack` (which combines any adapter and renderer)

Creating a custom renderer

What is a renderer

In JBrowse 1, a track type typically would directly call the data parser and do its own rendering. In JBrowse 2, the data parsing and rendering is offloaded to a web-worker or other RPC. This allows things to be faster in many cases. This is conceptually related to “server side rendering” or SSR in React terms.

How does jbrowse 2 use webworkers?

Figure 17: Conceptual diagram of how a track calls a renderer using the RPC

:::warning Note You can make custom track types that do not use this workflow, but it is a built-in workflow that functions well for the core track types in JBrowse 2, and is recommended. :::

How to create a new renderer

The fundamental aspect of creating a new renderer is creating a class that implements the “render” function. A renderer is actually a pair of a React component that contains the renderer’s output, which we call the “rendering”, and the renderer itself.

```java
class MyRenderer implements ServerSideRendererType {
    render(props) {
        const { width, height, regions, features } = props
        const canvas = createCanvas(width, height)
        const ctx = canvas.getContext('2d')
        ctx.fillStyle = 'red'
        ctx.drawRect(0, 0, 100, 100)
    }
```
const imageData = createImageBitmap(canvas)
return {
  reactElement: React.createElement(this.ReactComponent, { ...props }),
  imageData,
  height,
  width,
}
}

In the above simplified example, our renderer creates a canvas using width and height that are supplied via arguments, and draw a rectangle. We then return a `React.createElement` call which creates a “rendering” component that will contain the output.

:::info Note The above canvas operations use an `OffscreenCanvas` for Chrome, or in other browsers serialize the drawing commands to be drawn in the main thread. :::

What are the props passed to the renderer

The typical props that a renderer receives:

```typescript
export interface PileupRenderProps {
  features: Map<string, Feature>
  layout: { addRect: (featureId, leftBp, rightBp, height) => number }
  config: AnyConfigurationModel
  regions: Region[]
  bpPerPx: number
  height: number
  width: number
  highResolutionScaling: number
}
```

The layout is available on BoxRendererType renderers so that it can layout things in pileup format, and has an `addRect` function to get the y-coordinate at which to render your data.

The features argument is a map of feature ID to the feature itself. To iterate over the features Map, we can use an iterator or convert to an array:

```typescript
class MyRenderer extends ServerSideRendererType {
  render(props) {
    const { features, width, height } = props
    // iterate over the ES6 map of features
    for (const feature in features.values()) {
      // render each feature to canvas or output SVG
    }

    // alternatively
    const feats = Array.from(features.values())
    feats.forEach(feat => {})
  }
}
```

Adding custom props to the renderer

Track models themselves can extend this using their `renderProps` function.

For example, the `WiggleTrack` has code similar to this, which adds a `scaleOpts` prop that gets passed to the renderer:
const model = types
  .compose(
    'WiggleTrack',
    blockBasedTrack,
    types.model({
      type: types.literal('WiggleTrack'),
    }),
  )
  .views(self => {
    const { renderProps: superRenderProps } = self
    return {
      renderProps() {
        return {
          ...superRenderProps(),
          scaleOpts: {
            domain: this.domain,
            stats: self.stats,
            autoscaleType: getConf(self, 'autoscale'),
            scaleType: getConf(self, 'scaleType'),
            inverted: getConf(self, 'inverted'),
          },
        },
      },
    },
  });

Rendering SVG

Our SVG renderer is an example, where it extends the existing built in renderer type with a custom ReactComponent only:

```javascript
export default class SVGPlugin extends Plugin {
  install(pluginManager: PluginManager) {
    pluginManager.addRendererType(
      () =>
        new BoxRendererType({
          name: 'SvgFeatureRenderer',
          ReactComponent:SvgFeatureRendererReactComponent,
          configSchema: svgFeatureRendererConfigSchema,
          pluginManager,
        }),
      )
    }
  }
}
```

Then, we have our Rendering component just be plain React code. This is a highly simplified SVG renderer just to illustrate:

```javascript
export default function SvgFeatureRendering(props) {
  const { width, features, regions, layout, bpPerPx } = props
  const region = regions[0]
  const feats = Array.from(features.values())
  const height = readConfObject(config, 'height', { feature })
  return (<svg
```
{feats.map(feature => {
    // our layout determines at what y-coordinate to
    // plot our feature, given all the other features
    const top = layout.addRect(
        feature.id(),
        feature.get('start'),
        feature.get('end'),
        height,
    )
    const [left, right] = bpSpanPx(
        feature.get('start'),
        feature.get('end'),
        region,
        bpPerPx,
    )
    return <rect x={left} y={top} height={height} width={right - left} />
})
</svg>

:::info Note
1. The above SVG renderer is highly simplified, but it shows that you can have a simple React component that leverages the existing BoxRendererType, so that you do not have to necessarily create your own renderer class.

2. The renderers receive an array of regions to render, but if they are only equipped to handle one region at a time then they can select only rendering to regions[0].

**Overriding the renderer’s getFeatures method**

Normally, it is sufficient to override the getFeatures function in your dataAdapter. If you want to drastically modify the feature fetching behavior, you can modify the renderer’s getFeatures call.

The base ServerSideRendererType class has a built-in getFeatures function that, in turn, calls your adapter’s getFeatures function, but if you need tighter control over how your adapter’s getFeatures method is called, then your renderer.

The Hi-C renderer type does not operate on conventional features and instead works with contact matrices, so the Hi-C renderer has a custom getFeatures function:

```javascript
import { toArray } from 'rxjs/operators'
class HicRenderer extends ServerSideRendererType {
    async getFeatures(args) {
        const { dataAdapter, regions } = args
        const features = await dataAdapter.getFeatures(regions[0])
        .pipe(toArray())
        .toPromise()
        return features
    }
}
```

**Creating a custom track type**

Important note: track types are “high level concepts”, and don’t do a lot of logic: instead “display” types register themselves to a track type, to display data in a particular view. Then, the renderers are called by the display type on e.g. a per-block basis.
Examples of track types

For examples of custom track types, refer to things like:

- **HicTrack** - uses the `LinearHicDisplay` which calls the `HicRenderer` to draw contact matrix
- **VariantTrack** uses the `ChordVariantDisplay` and `LinearVariantDisplay` to draw itself differently in the `CircularView` and `LinearGenomeView` respectively. display type that allows a VCF to render “chords” in the `CircularView`. In the `LinearGenomeView`, the `LinearVariantDisplay` is quite similar to a normal feature track, but also has a custom feature details widget because it overrides the “selectFeature” behavior.
- **Synten track** - synten tracks can be displayed in multiple view types like DotplotView and LinearSyntenView. It uses the `DotplotDisplay` or `LinearSyntenDisplay` respectively to achieve this
Creating a custom view type

View types include things like our DotplotView, CircularView, SpreadsheetView, and LinearGenomeView. At their core, views are a “panel” that can show arbitrary contents.

The tutorial here shows how to create a simple “Hello view” that says “Hello world!”
Extension points

In the core codebase, we have the concept of extension points that users can call or add to.

The API is

```javascript
// extra props are optional, can pass an extra context object your extension point receives
pluginManager.evaluateExtensionPoint(extensionPointName, args, props)
```

There is also an async method:

```javascript
// extra props are optional, can pass an extra context object your extension point receives
pluginManager.evaluateAsyncExtensionPoint(extensionPointName, args, props)
```

Users can additionally add to extension points, so that when they are evaluated, it runs a chain of callbacks that are registered to that extension point:

```javascript
pluginManager.addToExtensionPoint(extensionPointName, callback => newArgs)
```

The newArgs returned by your callback are passed on as the args to the next in the chain.

Here are the extension points in the core codebase:

**Core-extendPluggableElement**

**args:**

- `pluggableElement: PluggableElement` - this
type: synchronous

used to add extra functionality to e.g state tree models, for example, extra right-click context menus. your callback will receive every pluggable element registered to the system

https://github.com/GMOD/jbrowse-components/blob/6ceeac51f8bcecf3b0a99e23f2277a6e5a7662e/plugins/dotplot-view/src/extensionPoints.ts#L9-L43

**Core-guessAdapterForLocation**

type: synchronous

used to infer an adapter type given a location type from the “Add track” workflow. you will receive a callback asking if you can provide an adapter config given a location object

https://github.com/GMOD/jbrowse-components/blob/6ceeac51f8bcecf3b0a99e23f2277a6e5a7662e/plugins/gff3/src/index.ts#L27-L53

**Core-guessTrackTypeForLocation**

type: synchronous

used to infer a track type given a location type from the “Add track workflow”

example https://github.com/GMOD/jbrowse-components/blob/6ceeac51f8bcecf3b0a99e23f2277a6e5a7662e/plugins/alignments/src/index.ts#L108-L118

**Core-extendSession**

type: synchronous

used to extend the session model itself with new features

- `session: AbstractSessionModel` - instance of the session model to customize the about dialog

48
Core-customizeAbout
type: synchronous

- config: Record<string,unknown> - a snapshot of a configuration object that is displayed in the about dialog

TrackSelector-multiTrackMenuItems
type: synchronous

used to add new menu items to the “shopping cart” in the header of the hierarchical track menu when tracks are added to the selection

example https://github.com/GMOD/jbrowse-components/blob/6ceeac5f8bcecf3b0a99e23f2277a6e5a7662e/plugins/wiggle/src/CreateMultiWiggleExtension/index.ts#L10-L67

LaunchView-LinearGenomeView
type: async

launches a linear genome view given parameters. it is not common to extend this extension point, but you can use it as an example to create a LaunchView type for your own view

- session: AbstractSessionModel - instance of the session which you can call actions on
- assembly: string - assembly name
- loc: string - a locstring
- tracks: string[] - array of trackIds

https://github.com/GMOD/jbrowse-components/blob/6ceeac5f8bcecf3b0a99e23f2277a6e5a7662e/plugins/linear-genome-view/src/index.ts#L131-L189

LaunchView-CircularView
type: async

similar to LaunchView-LinearGenomeView, this is not common to extend, but you can use it as an example to create a LaunchView type for your own view

- session: AbstractSessionModel - instance of the session which you can call actions on
- assembly: string - assembly name
- tracks: string[] - array of trackIds

https://github.com/GMOD/jbrowse-components/blob/6ceeac5f8bcecf3b0a99e23f2277a6e5a7662e/plugins/circular-view/src/index.ts#L30-L66

LaunchView-SvInspectorView
type: async

launches a sv inspector with given parameters. it is not common to extend this extension point, but you can use it as an example to create a LaunchView type for your own view

- session: AbstractSessionModel - instance of the session which you can call actions on
- assembly: string - assembly name
- uri: string - a url to load
- fileType?: string - type of file referred to by locstring (VCF|CSV|BEDPE, etc)

https://github.com/GMOD/jbrowse-components/blob/6ceeac5f8bcecf3b0a99e23f2277a6e5a7662e/plugins/sv-inspector/src/index.ts#L121-L61

LaunchView-SpreadsheetView
type: async


launches a sv inspector with given parameters. It is not common to extend this extension point, but you can use it as an example to create a LaunchView type for your own view

- **session**: AbstractSessionModel - instance of the session which you can call actions on
- **assembly**: string - assembly name
- **uri**: string - a url to load
- **fileType?**: string - type of file referred to by locstring (VCF|CSV|BEDPE, etc)

https://github.com/GMOD/jbrowse-components/blob/6ceeac51f8bcecfc3b0a99e23f2277a6e5a7662e/plugins/spreadsheet-view/src/index.ts#L26-L59

LaunchView-DotplotView

*type: async*

launches a dotplot with given parameters. It is not common to extend this extension point, but you can use it as an example to create a LaunchView type for your own view

- **session**: AbstractSessionModel - instance of the session which you can call actions on
- **views**: { loc: string; assembly: string; tracks?: string[] }[] - view params for vertical and horizontal
- **tracks**: string[] - trackIds to turn on

https://github.com/GMOD/jbrowse-components/blob/6ceeac51f8bcecfc3b0a99e23f2277a6e5a7662e/plugins/dotplot-view/src/LaunchDotplotView.ts#L7-L46

LaunchView-DotplotView

*type: async*

launches a linear synteny view with given parameters. It is not common to extend this extension point, but you can use it as an example to create a LaunchView type for your own view

- **session**: AbstractSessionModel - instance of the session which you can call actions on
- **views**: { loc: string; assembly: string; tracks?: string[] }[] - view params for vertical and horizontal
- **tracks**: string[] - trackIds to turn on

https://github.com/GMOD/jbrowse-components/blob/6ceeac51f8bcecfc3b0a99e23f2277a6e5a7662e/plugins/linear-comparative-view/src/LaunchLinearSyntenyView.ts#L9-L68

Core-replaceAbout

*type: synchronous*

adds option to provide a different component for the “About this track” dialog

- **session**: AbstractSessionModel - instance of the session which you can call
- **config**: AnyConfigurationModel - a configuration object for the track

Return value: The new React component you want to use

example: replaces about dialog for a particular track ID

```javascript
pluginManager.addToExtensionPoint(
  'Core-replaceAbout',
  (DefaultAboutComponent, { session, config }) => {
    return config.trackId === 'volvox.inv.vcf'
      ? NewAboutComponent
        : DefaultAboutComponent
  },
)
```
Core-extraAboutPanel

type: synchronous

type: synchronous

adds option to provide a different component for the “About this track” dialog

- **session**: AbstractSessionModel - instance of the session which you can call
- **config**: AnyConfigurationModel - a configuration object for the track

Return value: An object with the name of the panel and the React component to use for the panel

example: adds an extra about dialog panel for a particular track ID

```javascript
pluginManager.addToExtensionPoint(
  'Core-extraAboutPanel',
  (DefaultAboutExtra, { /* session, */ config }) => {
    return config.trackId === 'volvox_sv_test' ? {
      name: 'More info', Component: ExtraAboutPanel
    } : DefaultAboutExtra
  },
)
```

Core-customizeAbout

type: synchronous

- **config**: Record<string, unknown> a snapshot of a configuration object for the track, with formatAbout already applied to it

Return value: New config snapshot object

Core-replaceWidget

type: synchronous

adds option to provide a different component for the “About this track” dialog

- **session**: AbstractSessionModel - instance of the session which you can call
- **model**: WidgetModel - a widget model. This is called for every widget type, including configuration, feature details, about panel, and more. The feature details may be a common one. See Core-extraFeaturePanel also, matches the model attribute from there

Return value: The new React component you want to use

example: replaces about feature details widget for a particular track ID

```javascript
pluginManager.addToExtensionPoint(
  'Core-replaceAbout',
  (DefaultAboutComponent, { model }) => {
    return model.trackId === 'volvox.inv.vcf' ? NewAboutComponent : DefaultAboutComponent
  },
)
```

Note: it is not always possible to retrieve the configuration associated with a track that produced the feature details. Therefore, we check model.trackId that produced the popup instead. Note that if you want copies of your track to get same treatment, might use a regex to loose match the trackId (the copy of a track will have a timestamp and -sessionTrack added to it).
Core-extraFeaturePanel

type: synchronous

- **model**: `BaseFeatureWidget` - the `BaseFeatureWidget` model. This has properties `model.trackId`, `model.trackType`, and `model.track`, though track may be undefined if the user closed the track, while trackId and trackType will be defined even if user closed the track
- **feature**: `Record<string, unknown>` - a snapshot of a feature object
- **session**: `AbstractSessionModel` - instance of the session which you can call

Return value: An object with the name of the panel and the React component to use for the panel example

```javascript
pluginManager.addToExtensionPoint(
    'Core-extraFeaturePanel',
    (DefaultFeatureExtra, { model }) => {
        return model.trackId === 'volvox_filtered_vcf' ? {
            name: 'Extra info', Component: ExtraFeaturePanel
        } : DefaultFeatureExtra
    },
)
```

### Extension point footnote

Users that want to add further extension points can do so, by simply calling

```javascript
const returnVal = pluginManager.evaluateExtensionPoint(
    'YourCustomNameHere',
    processThisValue, extraContext,
)
```

Then, any code that had used:

```javascript
pluginManager.addToExtensionPoint(
    'YourCustomNameHere',
    (processThisValue, extraContext) => {
        /* the first arg is the "processThisValue" from the extension point, it may get mutated if multiple extension points are chained together

        the second argument to the extension point is the extra context from evaluating the extension point. it does not get mutated even if there is a chain of values, it is passed as is to each one*/
        return processThisValue
    },
)
```

The naming system, “Core-” just refers to the fact that these extension points are from our core codebase. Plugin developers may choose their own prefix to avoid collisions.
Pluggable elements

A plugin is an independently distributed package of code that is designed to “plug in” to a JBrowse application. It’s implemented as a class that extends @jbrowse/core/Plugin. It gets instantiated by the application that it plugs into, and it has an install method and a configure method that the application calls.

This class is distributed as a webpack bundle that exports it to a namespace on the browser’s window object specifically for JBrowse plugins. This means it is only possible to have one version of a particular plugin loaded on any given webpage, even if multiple products are loaded and using it on the same page.

It’s common for a plugin to use its configure method to set up mobx autoruns or reactions that react to changes in the application’s state to modify its behavior.

Plugins often also have their install method add “pluggable elements” into the host JBrowse application. This is how plugins can add new kinds of views, tracks, renderers, and so forth.

:::info Note Many of the plugins referenced in the following section are found in the JBrowse Github repo. We encourage you to reference and review the concepts presented here using the functional and up-to-date plugin code found there. :::

Pluggable elements

Pluggable elements are pieces of functionality that plugins can add to JBrowse. Examples of pluggable types include:

- Adapter types
- Track types
- View types
- Display types
- Renderer types
- Widgets
- RPC calls
- Extension points
- Internet account types
- Connection types
- Text search adapter types
- Extension points
- Add track workflow

In addition to creating plugins that create new adapters, track types, etc. note that you can also wrap the behavior of another track so these elements are composable.

For example, we can have adapters that perform calculations on the results of another adapter, views that contains other subviews, and tracks that contain other tracks, leading to a lot of interesting behavior.

Let’s dive further into these details, and look at some examples.

View types

Creating view types is one of the most powerful features of JBrowse 2, because it allows us to put entirely different visualizations in the same context as the standard linear-genome-view.

We have demonstrated a couple new view types in JBrowse 2 already, including:

- LinearGenomeView - the classic linear view of a genome
- CircularView - a Circos-like circular whole genome view
- DotplotView - a comparative 2-D genome view
- SvInspectorView - superview containing CircularView and SpreadsheetView subviews
- And more!

We think the boundaries for this are just your imagination, and there can also be interplay between view types e.g. popup dotplot from a linear view, etc.
Adapters

Adapters are parsers for a given data format. We will review what adapters the alignments plugin has (to write your own adapter, see creating adapters).

Example adapters: the @jbrowse/plugin-alignments plugin creates multiple adapter types:

- **BamAdapter** - This adapter uses the @gmod/bam NPM module, and adapts it for use by the browser.
- **CramAdapter** - This adapter uses the @gmod/cram NPM module. Note that CramAdapter also takes a sequenceAdapter as a subadapter configuration, and uses getSubAdapter to instantiate it.
- **SNPCoverageAdapter** - this adapter takes a BamAdapter or CramAdapter as a subadapter, and calculates feature coverage from it.

Track types

Track types are a high level type that controls how features are drawn. In most cases, a track combines a renderer and an adapter, and can do additional things like:

- Control what widget pops up on feature click
- Add extra menu items to the track menu
- Create subtracks (See AlignmentsTrack)
- Choose “static-blocks” rendering styles, which keeps contents stable while the user scrolls, or “dynamic-blocks” that update on each scroll

Example tracks: the @jbrowse/plugin-alignments exports multiple track types:

- **SNPCoverageTrack** - this track type actually derives from the WiggleTrack type
- **PileupTrack** - a track type that draws alignment pileup results
- **AlignmentsTrack** - combines SNPCoverageTrack and PileupTrack as “subtracks”

Displays

A display is a method for displaying a particular track in a particular view.

For example, we have a notion of a synteny track type, and the synteny track type has two display models:

- **DotplotDisplay**, which is used in the dotplot view
- **LinearSyntenyDisplay**, which is used in the linear synteny view

This enables a single track entry to be used in multiple view types e.g. if I run jbrowse add-track myfile.paf, this automatically creates a SyntenyTrack entry in the tracklist, and when this track is opened in the dotplot view, the DotplotDisplay is used for rendering.

Another example of a track type with multiple display types is VariantTrack, which has two display methods

- **LinearVariantDisplay** - used in linear genome view
- **ChordVariantDisplay** - used in the circular view to draw breakends and structural variants

Renderers

Renderers are a new concept in JBrowse 2, and are related to the concept of server side rendering (SSR), but can be used not just on the server but also in contexts like the web worker (e.g. the webworker can draw the features to an OffscreenCanvas). For more info see creating renderers.

For example, the @jbrowse/plugin-alignments exports several renderer types:

- **PileupRenderer** - a renderer type that renders Pileup type display of alignments fetched from the BamAdapter/CramAdapter
- **SNPCoverageRenderer** - a renderer that draws the coverage. Note that this renderer derives from the wiggle renderer, but does the additional step of drawing the mismatches over the coverage track

:::info Views, tracks, displays, renderers? If you’re confused about what kind of pluggable element you might need to accomplish your development goals, a way to remember the relationship between these four pluggable elements is as follows:
1. A view is a container for anything, views typically have tracks (the linear genome view especially)
2. A track controls the what (kind of data, data adapters used) and how (displays, renderers) of the data you’d like to display, typically within a view
3. A display is a way you might want to display the data on a track, you might have multiple displays for a given view, for example, displays can determine if a feature is drawn with rectangles or with triangles; displays may have renderers
4. A renderer controls how the display is presented, for example what might happen when you mouse over a feature

:::

Widgets

Widgets are custom info panels that can show up in side panels, modals, or other places in an app.

Widgets can do multiple types of things, including:

- Configuration widget
- Feature detail widget
- Add track widget
- Add connection widget
- etc.

These widgets can be extended via plugins, so for example, the @jbrowse/plugin-alignments extends the BaseFeatureDetailWidget to have custom display of the alignments.

- AlignmentsFeatureDetailWidget - this provides a custom widget for viewing the feature details of alignments features that customizes the basic feature detail widget

RPC methods

Plugins can register their own RPC methods, which can allow them to offload custom behaviors to a web-worker or server side process.

The wiggle plugin, for example, registers two custom RPC method types:

- WiggleGetGlobalStats
- WiggleGetMultiRegionStats

These methods can run in the webworker when available.

Add track workflows

Plugins can register their own React component to display in the “Add track” widget for adding tracks that require custom logic. The Multi-wiggle track is an example of this, it produces a textbox where you can paste a list of files.

A simple addition to the add track workflow:

```javascript
// plugins/wiggle/MultiWiggleAddTrackWidget/index.jsx

import PluginManager from '@jbrowse/core/PluginManager'
import { AddTrackWorkflowType } from '@jbrowse/core/pluggableElementTypes'
import { types } from 'mobx-state-tree'

// locals
import MultiWiggleWidget from './AddTrackWorkflow'

export default (pm: PluginManager) => {
  pm.addAddTrackWorkflowType(
    () =>
      new AddTrackWorkflowType({
        name: 'Multi-wiggle track',
```
/* in a separate file, export the react component to render within the track widget, typically a form to collect relevant data for your track */
ReactComponent: MultiWiggleWidget,
stateModel: types.model({}),
)

...and ensure you install this component into your larger plugin:

// plugins/wiggle/index.jsx
// ...

export default class WigglePlugin extends Plugin {
  name = 'WigglePlugin'

  install(pm: PluginManager) {
    // ...
    MultiWiggleAddTrackWidgetF(pm)
    // ...
  }
}

Extension points
Extension points are a pluggable element type which allows users to add a callback that is called at an appropriate time.
Checkout the full extension point API or an example for adding context menu items for more detailed information.
The basic API is that producers can say:

```javascript
const ret = pluginManager.evaluateExtensionPoint('ExtensionPointName', {
  value: 1,
})
```

And consumers can say:

```javascript
pluginManager.addToExtensionPoint('ExtensionPointName', arg => {
  return arg.value + 1
})
pluginManager.addToExtensionPoint('ExtensionPointName', arg => {
  return arg.value + 1
})
```

In this case, `arg` that is passed in evaluateExtensionPoint calls all the callbacks that have been registered by addToExtensionPoint. If multiple extension points are registered, the return value of the first extension point is passed as the new argument to the second, and so on (they are chained together).
So in the example above, `ret` would be `{value:3}` after evaluating the extension point.

Next steps
Now that you have an overview of the different pluggable element types that are available to you, review your understanding of the configuration model.
Also checkout the guided tutorial for writing a plugin, which will take you through everything from installation, creating a new pluggable element, and general development tips for working with JBrowse 2.
URL query parameter API

JBrowse Web features the ability to automatically provide URL parameters to setup a session.

Note that the embedded components like @jbrowse/react-linear-genome-view make no assumptions on how URL params are used, so would have to be implemented by the consumer of the library.

Simple API for linear genome view

We provide a simplified URL format specifically designed for launching a single linear genome view.

Example

http://host/jbrowse2/?config=test_data/config.json&loc=chr1:6000-7000&assembly=hg19&tracks=gene_track,vcf_track

Here are the query params used here:

?config=

Example

?config=test_data/volvox/config.json

A path to a JBrowse 2 config file, relative to the current folder on the disk. Note that this just uses client side fetch to read the file, not server side file reads. If ?config= is not specified, it looks for a file named config.json e.g. http://host/jbrowse2/config.json which is what the @jbrowse/cli tool sets up by default.

&assembly=

Example

&assembly=hg19

The &assembly parameter refers to an assembly’s “name” field one of the “assemblies” array in the from the config.json. This is only used for launching a single linear genome view.

&loc=

Example

&loc=chr1:6000-7000

This performs a navigation to this region on load, which can be specified using the syntax. This is only used for launching a single linear genome view.

Example strings

chr1:6000-7000 // using - notation for range
chr1:6000..7000 // using .. notation for range
chr1:7000 // centered on this position

Note: Navigating via a text search query e.g. supply &loc=gene_name is not yet supported.

&tracks=

Example

&tracks=gene_track,vcf_track

This is a comma separated list of trackIds. You can see your trackId’s in the config.json. Note, you can also refer to a trackId added by &sessionTracks= here. This is only used for launching a single linear genome view.
More URL parameters

&sessionTracks=

If you want to dynamically add a track to the session, you can do so with &sessionTracks=

You can also use this method to add a FromConfigAdapter track, which let’s you specify features in JSON format, so you can e.g. add BLAST hits via the URL bar

Example

uniqueId:one,refName:ctgA,start:190,end:191,name:Boris}]}&sessionTracks=[
  {
    "type": "FeatureTrack",
    "trackId": "url_track",
    "name": "URL track",
    "assemblyNames": ["volvox"],
    "adapter": {
      "type": "FromConfigAdapter",
      "features": [
        {
          "uniqueId": "one",
          "refName": "ctgA",
          "start": 190,
          "end": 191,
          "name": "Boris"
        }
      ]
    }
  }
]

&session=

The session parameter, e.g. &session= has a number of different “input formats”

Local sessions The local sessions look like this

https://host/jbrowse2/?session=local-Fjphq8kjY

By default, after a session is loaded, it is stored into localStorage, and then the URL bar uses the ?session=local-format to reflect the key of the localStorage entry.

Shared sessions If you click the “Share button” in the header bar, it will generate a “shareable link” that you can give to other users

https://host/jbrowse2/?session=share-HShsEcnq3i&password=nYzTU

See this FAQ entry for more info about how shared sessions work

Session spec Another useful session URL is called a “session spec” or “session specification”. This provides a way to launch multiple views at once, including view types other than the linear genome view
Linear Genome View

https://jbrowse.org/code/jb2/main/?config=test_data/volvox/config.json&session=spec-{"views":["LinearGenomeView","tracks":["gff3tabix_genes","volvox_filtered_vcf","volvox_microarray"]}

Live link
Expanded

```json
{
  "views": [
    {
      "assembly": "volvox",
      "loc": "ctgA:1-5100",
      "type": "LinearGenomeView",
      "tracks": [
        "gff3tabix_genes",
        "volvox_filtered_vcf",
        "volvox_microarray",
        "volvox_cram"
      ]
    }
  ]
}
```

As you can see, you can supply an array of views (so you can open multiple views at once) and can specify the loc, tracks, assembly, and view type, or other view specific parameters (different view types may accept different params, e.g. dotplot has two assemblies)

Circular view Here is a session spec for a Circular View

https://jbrowse.org/code/jb2/main/?config=test_data/volvox/config.json&session=spec-{"views":["CircularView","tracks":["volvox_sv_test"]}

Live link
Expanded

```json
{
  "views": [
    {
      "assembly": "volvox",
      "loc": "ctgA:1-5100",
      "type": "CircularView",
      "tracks": ["volvox_sv_test"]
    }
  ]
}
```

Dotplot view

https://jbrowse.org/code/jb2/main/?config=test_data/volvox/config_main_thread.json&session=spec-{"views":["DotplotView"]}

Live link
Expanded

```json
{
  "views": [
    {
      "type": "DotplotView",
      "tracks": ["volvox_fake_synteny"]
    }
  ]
}
```
Note that this dotplot session spec doesn’t have the ability to navigate to specific regions on the assembly yet, it just navigates to a whole genome overview.

**Spreadsheet view**

https://jbrowse.org/code/jb2/main/?config=test_data/volvox/config.json&session=%7B%22views%22:%5B%7B%22type%22:%22SpreadsheetView%22,%20%22uri%22:%22test_data/volvox/volvox.filterd.vcf.gz%22,%20%22assembly%22:%22volvox%22%7D%5D%7D

Live link
Expanded

```json
{
  "views": [
    {
      "type": "SpreadsheetView",
      "uri": "test_data/volvox/volvox.filterd.vcf.gz",
      "assembly": "volvox"
    }
  ]
}
```

**SV inspector**

https://jbrowse.org/code/jb2/main/?config=test_data/volvox/config.json&session=%7B%22views%22:%5B%7B%22type%3A%22SvInspectorView%22%2C%20%22uri%22:%22test_data/volvox/volvox.dup.vcf.gz%22%2C%20%22assembly%22:%22volvox%22%7D%5D%7D

Live link
Expanded

```json
{
  "views": [
    {
      "type": "SvInspectorView",
      "uri": "test_data/volvox/volvox.dup.vcf.gz",
      "assembly": "volvox",
    },
  ],
}
```

**Linear synten view**

https://jbrowse.org/code/jb2/main/?config=test_data%2Fvolvox%2Fconfig.json&session=%7B%22views%22:%5B%7B%22loc%22:%22ctgA:1-100%22%2C%20%22assembly%22:%22volvox%22%7D%2C%20%7B%22loc%22:%22ctgA:300-400%22%2C%20%22assembly%22:%22volvox%22%7D%5D%7D

Live link
Expanded

```json
{
  "views": [
    {
      "type": "LinearSyntenyView",
      "tracks": ["volvox_fake_synteny"],
      "views": [
        { "loc": "ctgA:1-100", "assembly": "volvox" },
        { "loc": "ctgA:300-400", "assembly": "volvox" }
      ]
    }
  ]
}
```
**JSON sessions**  Similar to encoded sessions, but more readable, JSON session let you specify the input a JSON snapshot of a session session. This is slightly different from a session spec, which has extra logic that loads the session. JSON sessions are literal session snapshots, like those that might come from the “Export session...” process

Example

```javascript
&session=json-"{"session":{"id":"xSHu7qGJN","name":"test","sessionPlugins":[]}}
```

This loads a session with an extra plugin loaded

**Encoded sessions**  This is similar to JSON sessions but uses a URL encoding (base64+gzip)

Example

```javascript
https://jbrowse.org/code/jb2/v1.5.9/?session=encoded-eJyNU2FzmkAQ_SvOfaaNIKDyLbF0x1rTWRq
```

Note that the “Share” button has a gear icon that let’s you select “Long URL” that produces these URLs. The encoded share links can be used without the central session sharing system in place, as the entire session is encoded in the URL.
API

This guide will cover the API level documentation for methods and tools useful to developers looking to enhance JBrowse or write plugins.

**MenuItems objects**

You can add menus or add items to existing menus in several places.

In these different places, a `MenuItem` object defines the menu item’s text, icon, action, and other attributes.

Types of `MenuItem`:

- **Normal**: a standard menu item that performs an action when clicked
- **Checkbox**: a menu item that has a checkbox
- **Radio**: a menu item that has a radio button icon
- **Divider**: a horizontal line (not clickable) that can be used to visually divide menus
- **SubHeader**: text (not clickable) that can be used to visually label a section of a menu
- **SubMenu**: contains menu items, for making nested menus

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Options are ‘normal’, ‘radio’, ‘checkbox’, ‘subMenu’, ‘subHeader’, or ‘divider’. If not provided, defaults to ‘normal’, unless a <code>subMenu</code> attribute is present, in which case it defaults to ‘subMenu’.</td>
</tr>
<tr>
<td>label</td>
<td>The text for the menu item. Not applicable to ‘divider’, required for all others.</td>
</tr>
<tr>
<td>subLabel</td>
<td>Additional descriptive text for the menu item. Not applicable to ‘divider’ or ‘subHeader’, optional for all others.</td>
</tr>
<tr>
<td>icon</td>
<td>An icon for the menu item. Must be compatible with Material-UI’s Icons. Not applicable to ‘divider’ or ‘subHeader’, optional for all others.</td>
</tr>
<tr>
<td>disabled</td>
<td>Whether or not the menu item is disabled (meaning grayed out and not clickable). Not applicable to ‘divider’ or ‘subHeader’, optional for all others.</td>
</tr>
</tbody>
</table>
| checked | Whether or not the checkbox or radio button are selected. Only applicable to ‘radio’ and ‘checkbox’.
- `onClickCallback` of action to perform on click. Function signature is `(session)=> undefined`. Required for ‘normal’, ‘radio’, and ‘checkbox’, not applicable to any others. |
| subMenu | An array of menu items. Applicable only to ‘subMenu’. |

As an example, here is an array of MenuItems and the resulting menu:

```javascript
;
{
  label: 'Normal menu item',
  icon: AddIcon,
  onClick: () => {},
},
{
  label: 'Normal',
  subLabel: 'with subLabel',
  icon: AddIcon,
  onClick: () => {},
},
{
  label: 'Disabled menu item',
  disabled: true,
  icon: AddIcon,
  onClick: () => {},
},
{
  type: 'radio',
  label: 'Radio checked',
```
checked: true,
onClick: () => {},
,
{  
type: 'radio',
label: 'Radio unchecked',
checked: false,
onClick: () => {},
},

{  
type: 'checkbox',
label: 'Checkbox checked',
checked: true,
onClick: () => {},
},

{  
type: 'checkbox',
label: 'Checkbox unchecked',
checked: false,
onClick: () => {},
},

{ type: 'divider' },
{ type: 'subHeader', label: 'This is a subHeader' },

{  
label: 'SubMenu',
SubMenu: [
   
   {  
      label: 'SubMenu item one',
      onClick: () => {},
   },
   
   {  
      label: 'SubMenu item two',
      onClick: () => {},
   },
   ],
}],
Figure 18: This screenshot shows all the various track menu options, generated by the code listing

rootModel Menu API

Users can customize the top-level menu items using these functions that are available on the rootModel:

appendMenu  Add a top-level menu

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuName</td>
<td>Name of the menu to insert.</td>
</tr>
</tbody>
</table>
Return Value  The new length of the top-level menus array

**insertMenu**  Insert a top-level menu

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuName</td>
<td>Name of the menu to insert.</td>
</tr>
<tr>
<td>position</td>
<td>Position to insert menu. If negative, counts from the end, e.g. <code>insertMenu('My Menu', -1)</code> will insert the menu as the second-to-last one.</td>
</tr>
</tbody>
</table>

Return Value  The new length of the top-level menus array

**appendToMenu**  Add a menu item to a top-level menu

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuName</td>
<td>Name of the top-level menu to append to.</td>
</tr>
<tr>
<td>menuItem</td>
<td>Menu item to append.</td>
</tr>
</tbody>
</table>

Return Value  The new length of the menu

**insertInMenu**  Insert a menu item into a top-level menu

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuName</td>
<td>Name of the top-level menu to insert into.</td>
</tr>
<tr>
<td>menuItem</td>
<td>Menu item to insert.</td>
</tr>
<tr>
<td>position</td>
<td>Position to insert menu item. If negative, counts from the end, e.g. <code>insertInMenu('My Menu', -1)</code> will insert the menu as the second-to-last one.</td>
</tr>
</tbody>
</table>

Return Value  The new length of the menu

**appendToSubMenu**  Add a menu item to a sub-menu

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuPath</td>
<td>Path to the sub-menu to add to, starting with the top-level menu (e.g. <code>['File', 'Insert']</code>).</td>
</tr>
<tr>
<td>menuItem</td>
<td>Menu item to append.</td>
</tr>
</tbody>
</table>

Return value  The new length of the sub-menu

**insertInSubMenu**  Insert a menu item into a sub-menu

Parameters
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuPath</td>
<td>Path to the sub-menu to add to, starting with the top-level menu (e.g. ['File', 'Insert']).</td>
</tr>
<tr>
<td>menuItem</td>
<td>Menu item to insert.</td>
</tr>
<tr>
<td>position</td>
<td>Position to insert menu item. If negative, counts from the end, e.g. <code>insertMenu('My Menu', -1)</code> will insert the menu as the second-to-last one.</td>
</tr>
</tbody>
</table>

**Return value**  The new length of the sub-menu
Installing and using plugins

The following will provide a short tutorial on how to use some of the many plugins available for JBrowse 2. Those plugins featured here are a part of the cancer-related database plugins on the cancer portal.

Prerequisites

- JBrowse 2 is installed on web or desktop
- Familiarity editing JSON
- Optionally, the JBrowse CLI installed

Installing plugins with one click

Some plugins provide all of their available functionality via the JBrowse interface, and thus can be fully accessed after utilizing the one-click install plugin store within the application.

![Figure 19: A screenshot of the plugin store. The ‘install’ button instantly installs the plugin. Plugins available via one-click install are shown the same on desktop and on web instances of JBrowse 2.](image)

GDC plugin

The GDC plugin adapts the resources available through NCI’s Genomic Data Commons.

The GDC plugin provides data adapters for the GDC API to retrieve cancer-related mutations and genes and for unique file types available via the GDC data store. The GDC plugin also provides a new internet account type that permits users to enter their GDC token to authenticate against protected data.

Adding the GDC plugin to a JBrowse session  The GDC plugin is fully accessible through the JBrowse interface. Utilize the plugin store’s one-click install, or add the plugin to your configuration file, like so:

```json
{
// ... plugins ...
```
Optionally, you can add a GDC Track to your configuration like so:

```json
{
    "type": "GDCTrack",
    "trackId": "gdc_plugin_track",
    "name": "GDC Explore",
    "assemblyNames": ["hg38"],
    "category": ["Annotation"],
    "adapter": {
        "GDCAdapterId": "DefaultGDCAdapter",
        "type": "GDCAdapter"
    },
    "displays": [
        {
            "type": "LinearGDCDisplay",
            "displayId": "gdc_plugin_track_linear",
            "renderer": {
                "color1": "jexl: cast({LOW: 'blue', MODIFIER: 'goldenrod', MODERATE: 'green', HIGH: 'red'}[get(feature, 'consequence').hits.edges[.node.transcript.is_canonical == true][0].node.transcript.annotation.vep_impact] || 'lightgray'",
                "labels": {
                    "name": "jexl: get(feature, 'genomicDnaChange')"
                },
                "type": "SvgFeatureRenderer"
            }
        }
    ]
}
```

However these track types are also available to be added to a session via the GDC data import panel.
The GDC data import panel  The data import panel provides several options to import data originating from the GDC into JBrowse.

If you have GDC files downloaded locally, you can drag and drop them into the first Drag and Drop panel to populate your JBrowse session with them.

You can also utilize an export functionality available via the GDC to bulk-import files if you drag that file into the panel. More detailed instructions are provided via the ‘more info’ button on the panel.
Figure 21: The import file via UUID or URL panel permits a user to enter either the entire URL of a file or exploration session, or simply the UUID of a file to populate it into JBrowse.
Figure 22: The final quick-add panel, as mentioned earlier, allows a user to add as many GDC Explore tracks as they want to a session. These tracks provide the user with access to the filterable track and datapoints retrieved through the GDC API.

**Adding files via the add track menu** Once installed, the track types and adapters provided by the GDC plugin are also available via the typical add track workflow.
How do I authenticate protected resources  Ensure that you have added the GDC internet account type to your configuration file. Presently, this must be done manually. Open the config file in a text processor and add the following:

```json
{
    "type": "GDCInternetAccount",
    "internetAccountId": "GDCExternalToken",
    "name": "GDC",
    "description": "GDC External Token",
    "domains": [
        "portal.gdc.cancer.gov",
        "api.gdc.cancer.gov"
    ],
    "customEndpoint": "http://localhost:8010/ proxy"
}
```

When you attempt to add a protected resource to your session either via the add track menu or the GDC data import panel, you will be prompted with a modal overlay to add your GDC token to the session. This token will be retained for other resources you add such that you will not have to add your token every time.
Filtering the GDC Explore track  When you have a GDC Explore track added to a session either via the config file or via the data import panel, you will be able to filter the features displayed on the track using the Filter option available through the track menu.

You can change whether you’re seeing mutations or genes using the topmost dropdown menu.

You can browse through the various filters via the tabs, and add new filters by clicking the ‘add’ button on the interface.

Changes made through the filter widget will populated automatically on the track it was opened for.

The bottom-most dropdown menu provides coloration options for the track, such that you can change the color in which features are represented based on a certain trait. A legend is provided to guide which color corresponds to which feature.
ICGC plugin

The ICGC plugin adapts the resources available through the International Cancer Genome Consortium.

Adding the ICGC plugin to a JBrowse session  The ICGC plugin is fully accessible through the JBrowse interface. Utilize the plugin store’s one-click install, or add the plugin to your configuration file, like so:

```javascript
{
    "plugins": [
    {
        "name": "ICGC",
        "url": "https://unpkg.com/jbrowse-plugin-icgc/dist/jbrowse-plugin-icgc.umd.production.js"
    }
]
}
```

Optionally, you can add an ICGC Track to your configuration like so:
However these track types are also available to be added to a session via the ICGC data import panel.

Figure 26: A screenshot of the ICGC data import panel opened to the right, and the access to the panel available through the tools menu. Like the GDC plugin, the ICGC plugin provides users with a quick-add button for adding additional ICGC tracks.

**Filtering the ICGC track** The instructions for how to use the filtering functionality of the ICGC track are essentially the same as those for the GDC Explore track. Please refer to those instructions.
Configuring plugins

Some plugins require their tracks to be configured via the config file because they access remote resources and do not otherwise provide a way to add these tracks to a session. After these plugins have been configured, you can use them with the appropriate assembly like any other track.

UCSC plugin

The UCSC plugin provides access to resources available through the UCSC API.

You can add the UCSC plugin to your JBrowse config file as follows:

```json
{
    "plugins": [
        {
            "name": "UCSC",
            "url": "https://unpkg.com/jbrowse-plugin-ucsc/dist/jbrowse-plugin-ucsc.umd.production.min.js"
        }
    ]
}
```

Then add the available tracks that are relevant to you under your “tracks” configuration, for example:

```json
{
    "type": "FeatureTrack",
    "trackId": "genehancer_ucsc_hg38",
    "name": "UCSC GeneHancer",
    "category": ["Annotation"],
    "assemblyNames": ["hg38"],
    "adapter": {
        "type": "UCSCAdapter",
```
Make sure to reference the many other available tracks in the UCSC example config

After adding one or more of these tracks to your configuration with the UCSC plugin installed, you will be able to freely enable them on linear genome views opened to the appropriate assembly using the track menu.

**Biothings plugin**

The Biothings plugin provides access to resources such as myvariant.info and mygene.info.

You can add the Biothings plugin to your JBrowse config file as follows:

```json
{
  "plugins": [
    {
      "name": "Biothings",
      "url": "https://unpkg.com/jbrowse-plugin-biothings/dist/jbrowse-plugin-biothings.umd.production.min.js"
    }
  ]
}
```

Then add the available tracks that are relevant to you under your “tracks” configuration, for example:

```json
{
  "type": "FeatureTrack",
  "trackId": "myvariant_hg19_evs",
  "name": "MyVariant v1 (evs)",
  "assemblyNames": ["hg19"],
  "category": ["Annotation"],
  "adapter": {
    "query": "query?q={ref}:{start}-{end} AND _exists_:evs & size=1000& size=1000& fields=gwassnps&email=colin.diesh@gmail.com",
    "baseUrl": "https://myvariant.info/v1/",
    "type": "MyVariantV1Adapter"
  }
}
```

Make sure to reference the many other available tracks in the Biothings example config

After adding one or more of these tracks to your configuration with the UCSC plugin installed, you will be able to freely enable them on linear genome views opened to the appropriate assembly using the track menu.

**CIVIC plugin**

The CIVIC plugin provides access to resources available through the CIVIC API.

You can add the CIVIC plugin to your JBrowse config file as follows:

```json
{
  "plugins": [
    {
      "name": "CIVIC",
      "url": "https://unpkg.com/jbrowse-plugin-civic/dist/jbrowse-plugin-civic.umd.production.min.js"
    }
  ]
}
```
Then add the relevant track to the “tracks” configuration:

```json
{
    "type": "FeatureTrack",
    "trackId": "ncbi_gff_hg19",
    "name": "NCBI RefSeq (GFF3Tabix)",
    "assemblyNames": ["hg19"],
    "category": ["Annotation"],
    "metadata": {
        "dateaccessed": "12/03/2020"
    },
    "adapter": {
        "type": "Gff3TabixAdapter",
        "gffGzLocation": {
            "uri": "https://s3.amazonaws.com/jbrowse.org/genomes/hg19/ncbi_refseq/GRCh37_latest_genomic.sort.gff.gz"
        },
        "index": {
            "location": {
                "uri": "https://s3.amazonaws.com/jbrowse.org/genomes/hg19/ncbi_refseq/GRCh37_latest_genomic.sort.gff.gz.tbi"
            }
        }
    }
}
```

Make sure to reference the many other available tracks in the CIVIC example config.

After adding one or more of these tracks to your configuration with the UCSC plugin installed, you will be able to freely enable them on linear genome views opened to the appropriate assembly using the track menu.
Figure 28: A screenshot of the UCSC, Biothings, and CIVIC tracks present in the add track menu for the assembly hg38 after adding them to your config.
Figure 29: A screenshot of the UCSC, Biothings, and CIVIC tracks present in the add track menu for the assembly hg19 after adding them to your config.

Conclusion
You should now have an understanding of how to add a plugin to a JBrowse session, and, if applicable, configure a track such that the adapter provided by the plugin is utilized.

Plugins such as those highlighted in this tutorial provide essential annotations and datasets to a user’s JBrowse session and as demonstrated can be accessed in a variety of ways.
Developing a simple JBrowse plugin

JBrowse 2 plugins can be used to add new pluggable elements (views, tracks, adapters, etc), and to modify behavior of the application by adding code that watches the application’s state.

For the full list of what kinds of pluggable element types plugins can add, see the pluggable elements page.

The following tutorial will walk you through establishing your developer environment, spinning up a plugin, and running a local JBrowse instance with your custom plugin functionality.

Prerequisites

- git
- A stable and recent version of node
- yarn or npm
- basic familiarity with the command line, React, package management, and npm

First we’re going to install and set up the project for development.

Use git to clone the plugin template

The easiest way to start developing your plugin for JBrowse 2 is to use the plugin template.

To clone the plugin template project, on the command line run:

```bash
# change jbrowse - plugin - my - project to whatever you wish
# clone https://github.com/GMOD/jbrowse-plugin-template.git
# change jbrowse-plugin-my-project
cd jbrowse-plugin-my-project
```

Initialize the project

To initialize your project run,

```bash
yarn init
```

You’ll be asked a few questions relating to your new project.

Most fields can be left blank, but make sure to enter a descriptive name for your plugin in the first field.

:::note
Tip A typical naming convention for JBrowse plugins is “jbrowse-plugin-”, or, if you are going to publish to an NPM organization, we advise “@myscope/jbrowse-plugin-.” :::

You also need to install the dependencies:

```bash
yarn # or npm i
```

Setup JBrowse 2

Finally, we’re going to run:

```bash
yarn setup
```

which will grab the latest release version of JBrowse 2 (in the .jbrowse directory) and make it easy for you to run within your plugin project.

To run JBrowse:

```bash
yarn browse
```

You should see something like the following:
yarn run v1.22.10
$ npm-run-all jbrowse:*
$ shx cp jbrowse_config.json .jbrowse/config.json
$ cross-var serve --listen $npm_package_config_browse_port .jbrowse

We still need to run the plugin though; we need both to be running to test our plugin.

Open a new tab in your terminal and navigate again to your plugin project, then we’re going to run our plugin:

cd jbrowse-plugin-my-project
yarn start

Now you can navigate to http://localhost:8999/, and see your running JBrowse instance!

:::info Note At this point, you must be running your plugin on port 9000 to see a running JBrowse instance, otherwise you will meet a screen asking you to configure your instance.

We can verify our plugin has been added to our JBrowse session by clicking the first square on the splash screen “Empty,” and then navigating Add -> Hello View in the menu bar. This is the example pluggable element that is added in the template plugin project.

Next, we’re going to add our own pluggable element to the plugin.

If you’re building a small plugin, or only want to develop against the latest release version.

However, if you:

Figure 30: Your browser should look something like the above screenshot.
• want to run your plugin on JBrowse desktop
• would like more descriptive stack traces and debugging
• want to test your plugin against the most recent developer build

You can develop against the latest JBrowse core build by taking a quick detour to our developing with JBrowse web and desktop tutorial.

Now that our environment is set up and running, we’re going to add our own pluggable element to the project and observe it running in JBrowse.

For this tutorial, we’re going to be creating a custom widget, and using a Jexl callback to open it when we click a chord on the circular genome view.

Add new files, stubs, and install dependencies

Add a new directory under `src` called `CircularViewChordWidget` with two files `CircularViewChordWidget.tsx`, and `index.tsx`.

This component is essentially just a react component we’re going to embed in a JBrowse widget.

A widget’s index.tsx

The index file is going to export what our `pluginManager` needs to recognize the widget: a `ReactComponent`, a `configSchema`, and a `stateModelFactory`.

```
circularViewChordWidget/index.tsx

import { ConfigurationSchema } from '@jbrowse/core/configuration'
import PluginManager from '@jbrowse/core/PluginManager'
import { ElementId } from '@jbrowse/core/util/types/mst'
import { types } from 'mobx-state-tree'
```
With mobx-state-tree, you can observe here we’re defining the properties of our widget and some actions it can take. Within the `model` method, we’re defining that the model of our `CircularViewChordWidget` has an `id`, a `type`, and `featureData`. We can define whatever we want in here. For example, we could add a “widgetByline” string property, and be able to use it later in our React component.

To add the new property and some functions:

```javascript
// ...
export function stateModelFactory(pluginManager: PluginManager) {
  const stateModel = types
    .model('CircularViewChordWidget', {
      id: ElementId,
      type: types.literal('CircularViewChordWidget'),
      featureData: types.frozen({}),
      widgetByline: 'Default widget byline', // NEW
    })
    .actions(self => ({
      setFeatureData(data: any) {
        self.featureData = data
      },
      clearFeatureData() {
        self.featureData = {}
      },
      // NEW
      setWidgetByline(byline: string) {
        self.widgetByline = byline
      },
      // NEW
      getWidgetByline() {
        return self.widgetByline
      },
    }));
    return stateModel
}
```
Within the `.actions` method, we’re defining methods for the model. These can be far more complex than just accessors and mutators (anything you want really).

If you have a particularly complex model, consider moving your component’s model into a `model.ts`, and then exporting the stateModel from `index.ts` similar to how the ReactComponent is exported.

**A widget’s ReactComponent**

Now that we have our model set up, let’s build a simple widget that will open when we click the circular genome view chord.

**CircularViewChordWidget.tsx**

```javascript
import React, { useState } from 'react'
import { observer } from 'mobx-react'

const CircularViewChordWidget = observer(({ model }: { model: any }) => {
  return (<div> </div>)
})

export default CircularViewChordWidget
```

It’s important to note the use of the `mobx` observer here: when making modifications to the model, you’ll see those changes populated in your widget, thanks to the observer. We export the widget such that it can be seen by `CircularViewChordWidget/index.tsx`.

We’ll make our widget do something basic: display the chord’s information and a message we can edit.

```javascript
import React from 'react'
import { observer } from 'mobx-react'
import TextField from '@material-ui/core'
import { FeatureDetails, BaseCard, BaseCard } from '@jbrowse/core/BaseFeatureWidget/BaseFeatureDetail'

const CircularViewChordWidget = observer(({ model }: { model: any }) => {
  // these are two properties we have in our model
  // widgetByline is going to start out as an empty string
  // but featureData will be populated with the information from our chord;
  // we'll talk about how that happens more later
  const { featureData, widgetByline } = model
  return (<div>
    {/**< features will always have a name, start, end, and id; they can have additional information too */}
    <BaseCard title={featureData.name}>
      {/* here we're just demonstrating using a basic property from the model and updating it with observer */}
      <h2>{widgetByline}</h2>
      <p>Care to change the widget byline?</p>
      <TextField
        onChange={(e: any) => model.setWidgetByline(e.target.value)}
      />
  </BaseCard>
</div>)
```

91
As noted in the codeblock, @jbrowse/core has some reusable UI components exported. If you see something proprietary in the application you’d like to reuse, investigate whether it’s exported by @jbrowse/core, and if not make a request for that component to be exported for use in your plugin.

Now that we have our component built, we can install it into our plugin and test it out.

**Install the plugin to JBrowse at runtime**

The file src/index.ts exports your plugin and installs all the necessary components to JBrowse at runtime such that it runs properly.

Your src/index.ts file is going to look something like the following right now:

```javascript
import Plugin from '@jbrowse/core/Plugin'
import PluginManager from '@jbrowse/core/PluginManager'
import { ViewType } from '@jbrowse/core/pluggableElementTypes'
import { SessionWithWidgets, isAbstractMenuManager } from '@jbrowse/core/util'
import { version } from '../package.json'
import {
  ReactComponent as HelloViewReactComponent,
  stateModel as helloViewStateModel,
} from './HelloView'

export default class SomeNewPluginPlugin extends Plugin {
  name = 'SomeNewPluginPlugin'
  version = version

  install(pluginManager: PluginManager) {
    pluginManager.addViewType(() => {
      return new ViewType({
        name: 'HelloView',
        stateModel: helloViewStateModel,
        ReactComponent: HelloViewReactComponent,
      })
    })
  }

  configure(pluginManager: PluginManager) {
    if (isAbstractMenuManager(pluginManager.rootModel)) {
      pluginManager.rootModel.appendToMenu('Add', {
        label: 'Hello View',
        onClick: (session: SessionWithWidgets) => {
          session.addView('HelloView', {})
        }
      })
    }
  }
}
You’ll notice we’re already adding a new view type and configuring the rootModel in the template’s project. We can use these patterns to add our widget.

code

```ts
// imports
// ...
import { ViewType, WidgetType } from '@jbrowse/core/pluggableElementTypes'
// notice we're importing the components we exported from
src/CircularViewChordWidget/index.ts
import {
  configSchema as circularViewChordWidgetConfigSchema,
  stateModelFactory as circularViewChordWidgetStateModelFactory,
  ReactComponent as CircularViewChordWidgetComponent
} from './CircularViewChordWidget'
// ...
install (pluginManager: PluginManager) {
  // ...
  pluginManager.addWidgetType(() => {
    return new WidgetType({
      name: 'CircularViewChordWidget',
      heading: 'Chord Details',
      configSchema: circularViewChordWidgetConfigSchema,
      stateModel: circularViewChordWidgetStateModelFactory(pluginManager),
      ReactComponent: CircularViewChordWidgetComponent,
    })
  })
}
// ...
```

This is also where we’ll add our Jexl callback function:

code

```ts
// ...
import { getSession } from '@jbrowse/core/util'
// ...
// Jexl callback functions are adding inside configure in the plugin class
configure(pluginManager: PluginManager) {
  // ...
  /* .jexl.addFunction is the method to add a function
  the first parameter is the name of your jexl function, and how you'll call it
  the second parameter is the supplementary properties the function needs, here, we need these three properties for
  the circular view's chord click function */
  pluginManager.jexl.addFunction('openWidgetOnChordClick',
    (feature: any, chordTrack: any) => {
      // the session contains a ton of necessary information about the
      // present state of the app, here we use it to call the
      // showWidget function to show our widget upon chord click
      const session = getSession(chordTrack)
      if (session) {
        // @ts-ignore
```
Now that we’ve configured the jexl function to our JBrowse session, we can use it essentially anywhere.

While we could programmatically tell certain displays to use this jexl function when they perform an action, for our use case (clicking a chord on the circular view), we can simply write it into our config file.

**Setup the configuration for proper testing**

To open a view in JBrowse, we need an assembly configured, append the following to your `jbrowse_config.json` file (i.e. after the “plugins” field):

```json
{
    "assemblies": [ 
        {
            "name": "hg38",
            "aliases": ["GRCh38"],
            "sequence": {
                "type": "ReferenceSequenceTrack",
                "trackId": "P6R5xbRqRr",
                "adapter": {
                    "type": "BgzipFastaAdapter",
                    "fastaLocation": {
                        "uri": "https://jbrowse.org.genomes/GRCh38/fasta/hg38.prefix.fa.gz",
                        "locationType": "UriLocation"
                    },
                    "faiLocation": {
                        "uri": "https://jbrowse.org.genomes/GRCh38/fasta/hg38.prefix.fa.gz.fai",
                        "locationType": "UriLocation"
                    },
                    "gziLocation": {
                        "uri": "https://jbrowse.org.genomes/GRCh38/fasta/hg38.prefix.fa.gz.gzi",
                        "locationType": "UriLocation"
                    }
                }
            },
            "refNameAliases": {
                "adapter": {
                    "type": "RefNameAliasAdapter",
                    "location": {
                        "uri": "https://jbrowse.org/genomes/GRCh38/hg38_aliases.txt"
                    }
                }
            }
        }
    ]
}
```
Take some time to dissect what’s being added here:

- we’re adding the assembly GRCh38
- it can be referenced either by its name (hg38) or its aliases (GRCh38)
- it has a sequence, which has a BgzipFastaAdapter from which the reference sequence is derived
- these FASTA’s are hosted on jbrowse.org, referenced as a UriLocation
- there is also a refNameAliases text file being used to derive the reference names of the assembly

We’re now going to add a track that will make use of our jexl function. As mentioned previously, you could add your jexl function programmatically to all tracks of this type, but for now we’re just adding it to our assembly for this specific track.

Take some time to dissect what’s being added here:

- this is a track that will appear in our track list when we run JBrowse against this assembly
- it’s a VariantTrack called “demo_vcf”
- it derives its data from a given UriLocation, the file is a .vcf file using the VcfAdapter
- it declares its display, the ChordVariantDisplay, and specifies its onChordClick callback function
- the specified onChordClick callback function is that which we defined in our plugin class, the jexl function
Figure 32: A screenshot of what it will look like when you add a track to your configuration; that is, it will be available in the add track menu when you open a view.

Testing it out

Run JBrowse with your new plugin and manually test

Everything is in place to test this widget we’ve added to the plugin project out.

If you shut your instance down, restart JBrowse and your plugin (yarn browse and yarn start).

Navigate to localhost:8999/?config=localhost:3000/jbrowse_config.json or equivalent to see JBrowse running with your config.

Now navigate:

1. Click Start a new session -> Empty
2. In the top menu bar, click Add -> Circular view
3. When the view is open, click Open beside the assembly
4. Click the far right three rows of rectangles icon in the top left of the circular view, Open track selector
5. Select the track we populated from our config
6. Click any chord in the circular view

Expected result:

The widget opens on the right hand side with two panels, one with our editable widget byline, and one with our feature data.
Figure 33: A screenshot of the widget displayed after clicking on the chord.

:::info Troubleshooting If you get to this point and note that nothing happens, open the developer tools in your browser and investigate the console errors. Also check your running process in your terminal for any errors. Review the code you added to ensure you didn’t miss any imports or statements. Check over your config file to ensure that “plugins”, “assemblies”, and “tracks” are all present for the configuration to work properly. :::

Writing a simple integration test with cypress

For completeness, we might want to write a few tests for our plugin to ensure that future changes we make do not break the application.

The jbrowse-plugin-template uses cypress to write its integration tests. For plugins, integration tests are a particularly good way to test functionality, as a failing test might indicate the plugin needs to be updated for a new version of JBrowse, or, if interfacing with a third-party API or toolset, that the plugin might have to be tweaked to suit these changes.

We’re going to write a simple integration test suite that executes the action we tested above.

Add a cypress test

See the cypress documentation for a dive into options and best practices for writing cypress tests. The following is a very brief overview to get you started.

Within the directory cypress, you’ll see a number of folders; you’ll likely only need to make use of fixtures and integration.

- **fixtures**: This directory is where you might place testing files or a testing config.json file.
- **integration**: This directory is where all your integration tests go. You can organize them however you want. Using the template project, there will be two in there already you can use as an example, but we’ll write one of our own.

Make your own circ_test.spec.ts file within cypress/integration, and populate it with the following to start:
describe('Circular chord widget tests', () => {
  it('can access the widget', () => {
    cy.exec('shx cp cypress/fixtures/hello_view.json .jbrowse')
    cy.visit('/?config=hello_view.json&session=spec-{"views": [{"type": "CircularView"}]},
    )
  })
})

Right now our test does two things:

- copies our fixture `hello_view.json` into `.jbrowse` and,
- visits our JBrowse URL (default configured to `localhost:8999`) with that configuration and a circular view open

Notice the use of URL params to speed up the test setup; using URL params like this can come in handy for larger suites.

Take a moment to add the track specification to `hello_view.json` for testing purposes:

```
"tracks": [
  {
    "type": "VariantTrack",
    "trackId": "demo_vcf",
    "name": "demo_vcf",
    "assemblyNames": ["hg38"],
    "category": ["Annotation"],
    "adapter": {
      "type": "VcfAdapter",
      "vcfLocation": {
        "locationType": "UriLocation",
        "uri": "https://s3.amazonaws.com/jbrowse.org/genomes/hg19/skbr3/reads_lr_skbr3.fa_ngmlr-0.2.3_mapped.sniffles1kb_auto_l8_s5_noalt.new.vcf"
      }
    }
  },
  "displays": [
    {
      "type": "ChordVariantDisplay",
      "displayId": "demo_ch_v_disp",
      "onChordClick": "jexl:openWidgetOnChordClick(feature, track, pluginManager)",
      "renderer": { "type": "StructuralVariantChordRenderer" }
    }
  ]
]
```

Running cypress

First, ensure both your plugin (yarn start) and JBrowse (yarn browse) are running. Open a third tab in your terminal at your project directory, and run yarn cypress:open. A cypress browser will open, click on the test you just wrote to run it.
Now that we have a visual on our test running, let’s add some more complexity:

```javascript
// ...
it('can access the widget', () => {
  // ...
  cy.contains('Open').click()
  cy.get('[data-testid="circular_track_select"]').click()
  cy.contains('demo_vcf').click()
  // if any asynchronous calls are made it might be pertinent to .wait
  cy.wait(2000)
  // we use force: true here to make sure we can click the chord, that's an
  // svg overlaid over many other svg's
  cy.get('[data-testid="chord-1591034956-148"]').click({ force: true })
  // we can see this text, so we know we've accomplished our goal
  cy.contains('Care to change the widget byline?')
})
```

We can add one more small test to check data input:

```javascript
it('can change the byline', () => {
  // you can use the share functionality to generate a session at any point
  // you might want to revisit for future tests
  cy.visit(`http://localhost:8999/?config=config.json&session=share-V0PG_1mjHJ&password=ho4Uq`,
    )
  cy.wait(2000)
  // .get can be used to nab elements of a certain type as well, here we're
  // referencing the third 'input'
  cy.get('input').eq(3).type('Some testing string')
  cy.get('input').eq(3).type('{enter}')
  cy.contains('Some testing string')
})
```
This test will check the functionality of our input field and updating the property on the widget model.

Run your suite again for completeness (you may have to reset your instance of cypress, then run `yarn cypress:open` again).

![Screenshot of cypress test running](image)

Figure 35: A screenshot of a cypress test running with the built-in browser; this is what your test should look like (and pass) when finished here.

**Next steps**

We have a complete and tested plugin, so now we’re ready to publish it to NPM and request that it be added to the plugin store.

Sometimes you might write a plugin that is specific to you or your organization’s needs, but you also might want to share it with the greater community. That’s where the plugin store shows off its strengths.

As a plugin developer, you can publish your plugin to NPM, and then request that your plugin be added to the plugin store. After your plugin is successfully whitelisted, you will see it within the JBrowse app’s plugin store widget and you and others can freely install the plugin into their JBrowse session. Any further publications you make to the plugin via NPM will automatically be updated for the plugin available through the plugin store.

The following document will describe how to accomplish this.

**Publish your plugin to NPM**

The following will guide you through publishing with NPM. You’ll need an NPM account and token to do this, so please set that up first through the NPM site.

If you’d prefer not to publish to NPM, you can host your plugin files elsewhere, just ensure the link is accessible publicly.
When your plugin is in a publishable state and you have NPM credentials, you can run the following within your plugin’s root directory (where `package.json` is found):

```bash
yarn publish
```

Set the version to whatever you’d like, enter your credentials, and then complete the publication process. Once you can see your package on NPM, move on to the next step.

**Request your plugin be added to the plugin store**

To populate your plugin to the plugin store, it must be added to the plugin list, a whitelist of JBrowse plugins. Navigate to the plugin list repository and use the github UI to **Fork** the repository.

![Repository](image)

**Comparing changes**

Choose two branches to see what’s changed or to start a new pull request. If you need to, you can also compare across forks.

---

**Figure 36:** Click the ‘Fork’ option at the top of the repository to create an editable clone of the repo.

:::info Tip
It’s easy enough to edit the files required using the github UI, but feel free to clone and push to the forked repo using your local environment as well.

:::

**Optional: create an image for your plugin**

An image helps communicate the capabilities of your plugin to adopters at a glance. Consider creating an 800 x 200 `.png` screenshot of a core feature of your plugin to show off.

We recommend using pngquant to compress your image to keep the repo manageable.

Once your image is all set, you can upload it to your forked repo (ideally in `~/jbrowse-plugin-list/img/`) using the Github UI or pushing the file from your computer.

**Adding the details for your plugin to the list**

Once forked, you can edit the `plugins.json` file to include the following information regarding your new plugin:

```json
{
  "plugins": [
    // ...other plugins already published,
    {
      // this plugin name needs to match what is in your package.json
```
Push your changes to the main branch of your forked repo when you’re done.

Make a pull request

Now that your plugin’s information is accurate, navigate again to the plugin list repository, and create a new pull request.

In the pull request UI, click “compare across forks” and select your fork as the head repository to merge into the main of jbrowse-plugin-list. Your changes should show in the editor, and you can create your PR.

Figure 37: Use the compare across forks option in the pull request UI to merge your forked repo’s main branch into the jbrowse-plugin-list main branch.

Next steps

The JBrowse development team will review your plugin to ensure that it is functional, then when it is merged in the plugin will be available on the plugin store.

In this tutorial, we set up a development environment for JBrowse 2 and added a custom pluggable element to a plugin.
We also published the plugin to NPM and requested that it be added to the JBrowse plugin store so others can access our plugin.

To learn more about the various pluggable elements available in JBrowse (and thus more that you can do with plugins!) checkout our developer guide documentation.

If you have further questions about plugin development, or development with JBrowse in general, stop by the JBrowse team gitter channel, or start a discussion on the jbrowse-components discussions forum.
Developing with JBrowse web and desktop

The following guide will walk you through setting up a developer environment for development with JBrowse web and JBrowse desktop.

This guide will provide some steps from the perspective of a plugin developer, but if you are interested in contributing to jbrowse-components, the setup steps are equally relevant.

Prerequisites

- git
- A stable and recent version of node
- yarn or npm
- basic familiarity with the command line, React, package management, and npm

Setup JBrowse 2 using the latest developer build

The code for both JBrowse web and JBrowse desktop is found in the jbrowse-components repository. First we’re going to clone the repo and install the dependencies.

```bash
git clone https://github.com/GMOD/jbrowse-components
cd jbrowse-components
yarn # or npm i
```

To run JBrowse web

```bash
cd products/jbrowse-web
yarn start # or npm run
```

JBrowse web will by default spin up on http://localhost:3000.

Configuration not found. You may have arrived here if you requested a config that does not exist or you have not set up your JBrowse yet.

Sample JBrowse configs:

- Volvox sample data
- Human basic
- Human sample data
- Tomato SVs
- Breakpoint
- Grape/Peach dotplot
- Grape/Peach synteny
- Yeast synteny
- Many configs
- Honeybee
- Wormbase
- WGR5 methylation

No config.json found. If you want to learn how to complete your setup, visit our quick start guide.

Figure 38: Screenshot of the ‘no config.json found’ screen of JBrowse web. This is what you will see when first spinning up JBrowse web.

You can select one of the sample configs to poke around with or to immediately start seeing changes you make in the codebase, or you can run JBrowse against a config with the plugin you’re developing. See the plugin tutorial if you need help starting with plugin development.

If you have a plugin running on port 9000 from the plugin tutorial, navigate to
and you’ll see the pluggable elements you’ve added through your plugin on your running JBrowse session.

Figure 39: If you haven’t made any changes to the plugin template, you’ll see ‘Hello view’ in the Add menu, as shown in this screenshot.

To run JBrowse desktop

Open two tabs in your terminal at ~/jbrowse-components; in one tab, run:

```
cd products/jbrowse-desktop
yarn serve
```

And in the other tab, run:

```
cd products/jbrowse-desktop
yarn develop
```

Doing this, you can quickly restart the “react app” part of JBrowse (the tab where you ran `yarn develop`) during your development without having to wait for the electron app (`yarn serve` tab) to restart, as you would if you simply ran `yarn start` in one tab.

A new window running JBrowse desktop will open.

Running JBrowse desktop with a plugin in development

The following assumes your plugin is running on port 9000, as shown in the plugin tutorial.
JBrowse will open on the splash screen when first spun up. The easiest way to see your local plugin running on JBrowse desktop is to select a quickstart assembly (under “Launch new session” on the left side of the screen) and then press “Go”.

![JBrowse splash screen]

Figure 40: The JBrowse desktop splash screen has some sample assemblies on the left panel.

Then navigate: **Tools -> Plugin Store** and press the button at the top of the Plugin Store widget “Add Custom Plugin.”
Figure 41: The ‘Add Custom Plugin’ button is at the top of the Plugin Store in JBrowse desktop.

It’s important to fill these fields in correctly, if you’ve followed the plugin tutorial, the information you need will be in the `jbrowse_config.json` file.

You might see something like the following in your `jbrowse_config.json` file:

```json
{
    "plugins": [
        {
            "name": "SomeNewPlugin",
            "url": "http://localhost:9000/dist/some-new-plugin.umd.development.js"
        }
    ]
}
```

To which you would fill the fields in like so:
Figure 42: After pressing the button a modal will open with some fields to fill in. Pictured above, your fields filled in might look like this. Make sure they match the information from your plugin project, that is currently running.

After pressing “Submit” on this form, your plugin should be added to your session. For easy access to this session, navigate **File -> Save as..** to save the `.jbrowse` file somewhere you can easily open it.

:::warning Note When developing your plugin using JBrowse desktop, the app will not automatically reload when you make changes to your plugin code. To see these changes applied, press **F5** to refresh the desktop react application.

You can also abort the running process under the “yarn develop” tab we set up earlier, and start it again. :::

If you would like to see your `console.log` output under the “yarn develop” tab of your desktop development environment, follow these steps:

1. Create or edit the `.bash_profile` file typically found in the root directory of your profile
2. Add `export ELECTRON_ENABLE_LOGGING=1` on a new line to the file
3. Restart your terminal and JBrowse `yarn develop` process

You should now be able to see console log statements in your terminal when developing on JBrowse desktop (denoted by a message prefixed with something like INFO:CONSOLE).

**Next Steps**

Now that you have your environments and your plugin running, you can start developing for JBrowse 2.

If you took a detour from the plugin tutorial, head back to where you left off.

If you’d like some general development information, checkout the series of developer guides available.
Writing a no-build plugin

The following guide will provide a short tutorial on how to create a single page no-build plugin for JBrowse 2.

Prerequisites

- you can run an instance of JBrowse 2 on the web, see any of our quickstart guides for details
- a stable and recent version of node
- basic familiarity with the command line and navigating the file system

What is the difference between a no-build plugin and a regular plugin?

A “regular” JBrowse plugin often uses our plugin template https://github.com/GMOD/jbrowse-plugin-template which uses rollup to compile extra dependencies that your plugin might use. In contrast, “no-build” plugins have no build step and can be hand edited. This can be useful for adding extra jexl config callbacks for making extra config callbacks or similar modifications.

Writing a no-build plugin

Adding a callback function which you can use in your config

A common method for a no-build plugin might be making a custom function that you can use to simplify jexl callbacks in your config. We will create a file myplugin.js, which will contain a “UMD” module providing a single “Plugin” class [1].

myplugin.js

```javascript
// the plugin will be a simplified UMD module, and we put the code in a function to avoid variable name collisions with the global scope
;(function () {
    class Plugin {
        name = 'MyNoBuildPlugin'
        version = '1.0'

        install(pluginManager) {
            pluginManager.jexl.addFunction('customColor', feature => {
                if (feature.get('type') === 'exon') {
                    return 'red'
                } else if (feature.get('type') === 'CDS') {
                    return 'green'
                }
            })
        }

        configure(pluginManager) {}
    }

    // the plugin will be included in both the main thread and web worker, so install plugin to either window or self (webworker global scope)
    (typeof self !== 'undefined' ? self : window).JBrowsePluginMyNoBuildPlugin = {
        default: Plugin,
    }
})();
```

Put this file myplugin.js in the same folder as your config file, and then, you can refer to this plugin and the custom function you added in your config.
Note that you can also provide an ESM module that has just `export default class` but this is not supported by all browsers, notably firefox, which cannot import ESM files in webworkers, so for maximum compatibility, we show are using the UMD format still. Once firefox gains support for ESM modules, we will update this!

**Adding a global menu item**

Another example of a no-build plugin is to add menu items or minor extension points. Here, we’re going to add a menu item using the `configure` method in the plugin class.

`myplugin.js`

```javascript
// ...
configure(pluginManager) {
  // adding a new menu to the top toolbar
  pluginManager.rootModel.insertMenu('Citations', 4)

  // appending a menu item to the new menu
  pluginManager.rootModel.appendToMenu('Citations', {
    label: 'Cite this JBrowse session',
    onClick: (session) => { }
  })
}
// ...
```
Importing with jbrequire

Because our plugin is not going to be built with any dependencies, the process for referencing external libraries is a little different.

If a package you need to use is found within the JBrowse core project, a special function `jbrequire` can provide your plugin access to these packages. Click here for a full list of packages accessible through `jbrequire`. Using `jbrequire` might look like this:

```js
const { types } = pluginManager.jbrequire('mobx-state-tree')
```

which would provide the functionality of mobx-state-tree through that value.

Our final no-build plugin looks as follows:

```js
export default class MyNoBuildPlugin {
    name = 'MyNoBuildPlugin'
    version = '1.0'

    install(pluginManager) {
        // here, we use jbrequire to reference packages exported through JBrowse
        const { ConfigurationSchema } = pluginManager.jbrequire(
            '@jbrowse/core/configuration',
        )
        const WidgetType = pluginManager.jbrequire(
            '@jbrowse/core/pluggableElementTypes/WidgetType',
        )
        const { ElementId } = pluginManager.jbrequire(
            '@jbrowse/core/util/types/mst',
        )
        const { types } = pluginManager.jbrequire('mobx-state-tree')
        const React = pluginManager.jbrequire('react')

        // this is our react component
        const CiteWidget = props => {
            // React.createElement can be used to add html to our widget component
            const header = React.createElement(
                'h1',
                null,
                'Cite this JBrowse session',
            )
            const content = React.createElement(
                'p',
                null,
            )

            return React.createElement('div', null, [header, content])
        }

        // we're adding a widget that we can open upon clicking on our menu item
        pluginManager.addWidgetType(() => {
            // adding a widget to the plugin
            return new WidgetType(
                name: 'CiteWidget',
            )
        })
    }
}
```
With JBrowse running and your plugin added to your config, your JBrowse session should look like the following:

![Screenshot of a running JBrowse instance with the simple no build plugin added. Note our top level menu item has been added, and upon clicking it our widget opens.](image)

**Conclusion and next steps**

Congratulations! You built and ran a single file no-build plugin in JBrowse.

If you’d like some general development information, checkout the series of developer guides available.

Have some questions? Contact us through our various communication channels.
Writing a desktop specific plugin

The following tutorial will demonstrate making a desktop-specific JBrowse plugin to run external tools such as bcftools and tabix on a VCF file.

Prerequisites

- git
- A stable and recent version of node
- yarn or npm
- basic familiarity with the command line, React, package management, and npm
- bcftools installed brew install bcftools or sudo apt install bcftools
- tabix installed brew install htslib or sudo apt install htslib

Why might I need a desktop-specific plugin

If you’ve developed a plugin before, you know that much of JBrowse 2’s functionality can operate across both web and desktop instances of JBrowse. The main difference between the two applications has to do with JBrowse desktop’s ability to run jobs on the machine it is running on, where web instances are beholden to the browser and the resources provided to it as a traditional web application.

JBrowse desktop utilizes electron to distribute the Javascript application as a cross-platform desktop application. You might decide to build a desktop specific plugin if you need additional resources that are not distributed over the web to process data, if you need to run CLI operations, or otherwise run machine-specific tools through JBrowse.

Writing a desktop-specific plugin

In this tutorial, we are going to write a plugin that performs some of the file preparation steps outlined in the CLI tutorial. Specifically, our plugin will provide users with an interface to submit their file, the simple script will accept the file, then the script will run the appropriate operations against it, and then it will provide JBrowse with the files in an acceptable format. This kind of workflow can only be done on JBrowse desktop.

A completed version of this plugin can be found here if you get stuck following along.

:::info Note Presently there is no formal procedure to differentiate desktop-specific plugins from universal plugins. Please include this information in the README of your repository if making a public-facing plugin, and follow the outlined steps for restricting plugin behaviour to desktop only. :::

Setup

Use git to clone the plugin template

The easiest way to start developing your plugin for JBrowse 2 is to use the plugin template.

To clone the plugin template project, on the command line run:

```sh
# change jbrowse-plugin-my-project to whatever you wish
git clone https://github.com/GMOD/jbrowse-plugin-template.git jbrowse-plugin-my-project
cd jbrowse-plugin-my-project
```

Initialize the project

To initialize your project run,

```sh
yarn init
```

You’ll be asked a few questions relating to your new project.

Most fields can be left blank, but make sure to enter a descriptive name for your plugin in the first field.

:::note Tip
A typical naming convention for JBrowse plugins is “jbrowse-plugin-”, or, if you are going to publish to an NPM organization, we advise “@myscope/jbrowse-plugin-”.

You also need to install the dependencies:

```bash
yarn # or npm i
```

**Create a .jbrowse configuration file**  
To make things easier for testing, you’re encouraged to make a simple .jbrowse configuration file.

It should look something like this:

```json
myconfig.jbrowse
{
  "configuration": {
    "rpc": {
      "defaultDriver": "WebWorkerRpcDriver",
      "drivers": {
        "MainThreadRpcDriver": {},
        "WebWorkerRpcDriver": {}
      }
    },
    "logoPath": {
      "locationType": "UriLocation",
      "uri": ""
    }
  },
  "plugins": [
    {
      "name": "DesktopTools",
      "url":
        "http://localhost:9000/dist/jbrowse-plugin-desktop-tools.umd.development.js"
    }
  ],
  "assemblies": [
    {
      "name": "volvox",
      "aliases": ["vvx"],
      "sequence": {
        "type": "ReferenceSequenceTrack",
        "trackId": "volvox_refseq",
        "metadata": {
          "date": "2020-08-20"
        },
        "adapter": {
          "type": "TwoBitAdapter",
          "twoBitLocation": {
            "uri": "http://localhost:9000/volvox.2bit",
            "locationType": "UriLocation"
          }
        }
      },
      "refNameAliases": {
        "adapter": {
          "type": "FromConfigAdapter",
          "adapterId": "W6DyPGJ0UU",
        }
      }
    }
  ]
}
```
Ensure you change the assembly to one that coordinates with your own files, or use our testing file (unzip it to get the raw vcf) with the configuration as shown above. You can download the volvox sample assembly and place it in your root directory of your plugin running at localhost:9000.

Place your myconfig.jbrowse file somewhere convenient; we’ll open it later.

**Run JBrowse Desktop** Follow the developing with JBrowse web and desktop tutorial to create a running instance of JBrowse 2 desktop. The tutorial will also have you run your template plugin on the app to test.

**Running CLI operations through JBrowse desktop**

**Creating the UI** We’ll need some way to accept the file from the user through the JBrowse UI. Adding a new widget here is the same as doing it for a typical JBrowse plugin (see the simple JBrowse 2 plugin guide if you haven’t already).

1. Create a basic component and add it through the plugin manager

```
import React from 'react'
import { AddTrackModel } from '@jbrowse/plugin-data-management'

export default function MyToolWidget({ model }) {
  // a very simple component to display for now
  return (
    <div>
      <p>
        This desktop plugin widget will allow you to submit an unindexed `{` vcf `}` file to JBrowse which will then use a script to sort, index, and zip the file. This circumvents any CLI operations you may have otherwise needed to do!
      </p>
    </div>
  )
}
```

```
import PluginManager from '@jbrowse/core/PluginManager'
import { AddTrackWorkflowType } from '@jbrowse/core/pluggableElementTypes'
import { types } from 'mobx-state-tree'
```
// locals
import MyToolWidget from './MyToolWidget'

export default (pluginManager: PluginManager) => {
  // this method of the pluginManager adds a dropdown menu item to the add track workflow
  // utilizing this method is useful for adding unique workflows to 'add track'
  pluginManager.addAddTrackWorkflowType(() =>
    new AddTrackWorkflowType({
      name: 'Raw variant track',
      ReactComponent: MyToolWidget,
      stateModel: types.model({}),
    }),
  );
}

2. Add the widget to your plugin's index.ts

./src/index.ts

import Plugin from '@jbrowse/core/Plugin'
import PluginManager from '@jbrowse/core/PluginManager'
import { version } from '../package.json'
import { isElectron } from '@jbrowse/core/util'
import MyToolWidgetF from './MyToolWidget'

export default class DesktopToolsPlugin extends Plugin {
  name = 'DesktopToolsPlugin'
  version = version

  install(pluginManager: PluginManager) {
    // this method "isElectron" is very important for developing a desktop plugin
    if (isElectron) {
      MyToolWidgetF(pluginManager)
    }
  }

  configure() {}
}

It is critical to include the isElectron check before installing your plugin. This method verifies that the process running the application is an electron app (i.e. desktop), and our plugin will (eventually) crash JBrowse if it is attempted to be run on web. By using this method where necessary, we ensure that if an individual runs JBrowse with a config file that includes this plugin on JBrowse web, the application will simply not install the plugin.

Try running your plugin as-is now:

yarn start your plugin, then, in your running instance of JBrowse desktop, navigate: Open > select your myconfig.jbrowse file.

If you navigate Add > Linear genome view > Open > Open track selector > + icon button > Add track you should see your simple message display after selecting the “Raw variant track” option in the dropdown.

We'll add some more UI elements that allow us to do something more interesting:

./src/MyToolWidget/MyToolWidget.tsx
import React, { useEffect, useState } from 'react'
import { Box, Button, FormControl, Paper, TextField, Typography } from '@mui/material'
import { makeStyles } from 'tss-react/mui'
import { getSession } from '@jbrowse/core/util'
import { AddTrackModel } from '@jbrowse/plugin-data-management'
import { LocalPathLocation, FileLocation, BlobLocation } from '@jbrowse/core/util/types'
import { getBlob, storeBlobLocation } from '@jbrowse/core/util/tracks'
import { isElectron } from '@jbrowse/core/util'

// adding some styling to our components
const useStyles = makeStyles()(theme => ({
  textbox: {
    width: '100%',
  },
  paper: {
    margin: theme.spacing(),
    padding: theme.spacing(),
  },
  submit: {
    marginTop: 25,
    marginBottom: 25,
    display: 'block',
  },
  filename: {
    marginLeft: theme.spacing(1),
  },
}))

// our widget component will need to be updated
export default function MyToolWidget({ model }: { model: AddTrackModel }) {
  const { classes } = useStyles()
  // controls the name state based on the file populated
  const [trackName, setTrackName] = useState(
    model.trackName !== null ? model.trackName : `myvcf - ${Date.now()}`,
  )

  // changes the name state based on the file populated
  useEffect(() => {
    setTrackName(model.trackName)
  }, [model.trackData])

  return (
    { /* formatting using paper and other mui elements */ }
    <Paper className={classes.paper}>
      <p>
        This desktop plugin widget will allow you to submit an unindexed {'.vcf'} file to JBrowse which will then use a script to sort, index, and zip the file. This circumvents any CLI operations you may have otherwise needed to do!
      </p>
    </Paper>
  )
}
function isBlobLocation(location: FileLocation): location is BlobLocation {
    return 'blobId' in location
}

// we're using a custom file chooser to ensure the user can only select local files
function LocalFileChooser(props: {
    location?: FileLocation
    setLocation: Function
    setName: Function
}) {
    const { classes } = useStyles()
    const { location, setLocation, setName } = props
    const [filename, setFilename] = useState('')

    const needToReload = location && isBlobLocation(location) && !getBlob(location.blobId)
    return (<Box display="flex" flexDirection="row" alignItems="center">
        <Box>
            <FormControl fullWidth>
                <Button variant="outlined" component="label">
                    Choose File
                    <input key="bcftools-filechooser" type="file" hidden onChange={({ target }) => {
                        const file = target && target.files && target.files[0]
                        if (file) {
                            if (isElectron) {
                                // here we are setting the location information for the file selected
                                // these features are necessary for the vcftabixadapter
                                setLocation({
                                    location,
                                    name: file.name
                                })
                            }
                        }
                    }} />
                </Button>
            </FormControl>
        </Box>
    </Box>)
}
Above, we added a component that allows us to select a local file for processing. Its information is propagated, and we have access to its data through the model.

In the `onClick` method of our Submit button, we want to execute our CLI operations.

**Setting up the script** To run a simple script, we're going to import the 'child_process' module from node:

```
// src/MyToolWidget/MyToolWidget.tsx
```

```typescript
// ... // place these somewhere above your components, below the imports // we need to ensure we're running on electron to load in this node package const exec = isElectron ? require('child_process').exec : () => {} // runs the exec function function execute(command: any, callback: any) {
  exec(command, (error: any, stdout: any, stderr: any) => {
    callback(stdout)
  })
}
```

Now, we can provide a command through the command parameter after the “Submit” button is clicked on our UI:

```
// src/MyToolWidget/MyToolWidget.tsx
```

```typescript
// ...
```
onClick={() => {
  // a small utility function for error checking
  function isLocalPathLocation(
    location: FileLocation,
  ) : location is LocalPathLocation {
    return 'localPath' in location
  }
  // we can determine the localpath from the data provided from the user
  // submitting a file through the file selector
  // @ts-ignore
  const localPath = isLocalPathLocation(model.trackData)
  ? model.trackData.localPath :
    ''
  // a simple command to be executed, showing us our localPath
  execute(
    `echo ${localPath}`
    (output: any) => {
      console.log(output)
    },
  )
}}
// ...

You can see how easy the execute function is to use now. Where we currently have “echo ${localPath}” is where we’re going to write our CLI commands.

If relevant when writing your own plugin, you can put any string of valid CLI commands here and your child process will execute them to the best of its ability.

Try refreshing your desktop instance and get the script to echo the localpath of your selected file.

Adding functionality to the script  For this plugin, the script we will run against the plain .vcf files provided through the JBrowse desktop UI will look like the following:

```bash
bcftools sort "myfile.vcf" > "myfile.vcf.sorted.vcf"
bgzip "myfile.vcf.sorted.vcf"
tabix "myfile.vcf.sorted.vcf.gz"
```

Remember to install these libraries (detailed in the prerequisites) before running your plugin operation. If a CLI operation cannot be executed, simply nothing happens.

```bash
./src/MyToolWidget/MyToolWidget.tsx
```

```
onClick={() => {
  // we've updated our execute command to run our three CLI commands one after another
  execute(
    `bcftools sort "$\{localPath\}" > "$\{localPath}.sorted.vcf" && bgzip
    "$\{localPath}.sorted.vcf" && tabix "$\{localPath}.sorted.vcf.gz"`,
    () => {
      // in our callback function, we process the data that has been populated
      model.setTrackData(
        localPath: `$\{localPath}.sorted.vcf.gz`,
        locationType: 'LocalPathLocation',
      )
    },
  )
}}
// ...
```
Testing the plugin

Running the plugin on JBrowse desktop

We have all the necessary infrastructure to run our plugin completely now. If you haven’t been testing and running your plugin along with the tutorial, remember you can run it with the following steps:

```bash
yarn serve # in the `~/.browse-components/products/jbrowse-desktop` directory

yarn develop # in another terminal tab in the same `~/.browse-components/products/jbrowse-desktop` directory

yarn start # in another terminal tab in your plugin `~/.browse-plugin-my-plugin/` directory, or similar
```

Then open your .jbrowse configuration file containing your plugin details, and navigate to the Add track workflow. Try using our volvox raw vcf file and volvox assembly for testing if you do not have your own data.

Your final product should look something like the following:
Distributing the plugin

Distributing a desktop plugin is very similar to the publishing steps outlined in the simple plugin tutorial. As stated before, we encourage you to clearly indicate when a plugin is intended for desktop use only, as desktop-only operations are likely to crash when used on a web application.

Next steps

After this tutorial you should have a general understanding of the capabilities and potential desktop-only workflows that can be developed for a plugin for JBrowse 2.

It should be reiterated that the functionality outlined in this tutorial is a simple example to get you started with working on more complicated workflows with more tools – virtually anything you might do with a script or on the command line can be integrated into the JBrowse UI to streamline a workflow, for example, creating BigWig files from BAM/CRAM files with mosdepth, gunzip, and bedGraphToBigWig tools.

If you’d like some general development information, checkout the series of developer guides available.

Core takeaways

- developing a plugin specifically for desktop can allow you to run operations locally or on a private process through JBrowse to assist in data display or development
- the `isElectron` function exported by `@jbrowse/core/util` is critical in identifying codeblocks that can only be executed when running on JBrowse
Embedded components

Our embedded components allow you to use individual JBrowse views in your application

JBrowse React Linear Genome View

This component consists of a single JBrowse 2 linear view.

- [@jbrowse/react-linear-genome-view](https://www.npmjs.com/package/@jbrowse/react-linear-genome-view) linear genome view React component on NPM
- Storybook - docs for the linear genome view React component

Here is a table of different usages of the [@jbrowse/react-linear-genome-view](https://www.npmjs.com/package/@jbrowse/react-linear-genome-view) using different bundlers

<table>
<thead>
<tr>
<th>Bundler</th>
<th>Source code</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>create-react-app v4</td>
<td>source code (download)</td>
<td>no polyfills needed in create-react-app v4. On newer versions of node, you can need to use <code>export NODE_OPTIONS=--openssl-legacy-provider</code> before building cra4 apps</td>
</tr>
<tr>
<td>create-react-app v5</td>
<td>source code (download)</td>
<td>for create-react-app v5, we use craco to update the webpack config to polyfill some node modules. This demo also uses webworkers, which is a unique ability with webpack 5. See <a href="https://jbrowse.org/storybook/lgv/main/?path=/story/linear-view--with-web-worker">https://jbrowse.org/storybook/lgv/main/?path=/story/linear-view--with-web-worker</a> for details</td>
</tr>
<tr>
<td>vite demo</td>
<td>source code (download)</td>
<td>for vite, we use rollup to polyfill some node polyfills similar to craco in create-react-app v5. Note, may not work with newly released vite 3.x, works in dev but fails in production so this example uses vite 2.x</td>
</tr>
<tr>
<td>next.js demo</td>
<td>source code (download)</td>
<td>uses next.js 12. Also see next.config.js to update basePath as needed</td>
</tr>
<tr>
<td>vanilla.js demo</td>
<td>source code (download)</td>
<td>uses a script tag to include a UMD bundle, and doesn’t require any transpilation or bundling. See also dev tutorial here <a href="https://jbrowse.org/jb2/docs/tutorials/embed_linear_genome_view/01_introduction/">https://jbrowse.org/jb2/docs/tutorials/embed_linear_genome_view/01_introduction/</a></td>
</tr>
</tbody>
</table>

JBrowse React Circular Genome View

This component consists of a single JBrowse 2 circular view.

- [@jbrowse/react-circular-genome-view](https://www.npmjs.com/package/@jbrowse/react-circular-genome-view) circular genome view React component on NPM
- Storybook - docs for the circular genome view React component

Here is a table of different usages of the [@jbrowse/react-circular-genome-view](https://www.npmjs.com/package/@jbrowse/react-circular-genome-view) using different bundlers

<table>
<thead>
<tr>
<th>Syntax Demo</th>
<th>Source code</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>create-react-app v4</td>
<td>source code (download)</td>
<td>no polyfills needed in create-react-app v4. On newer versions of node, you can need to use <code>export NODE_OPTIONS=--openssl-legacy-provider</code> before building cra4 apps</td>
</tr>
<tr>
<td>SyntaxDemo</td>
<td>Source code</td>
<td>Note</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>create-demo react-app v5</td>
<td>source code (download)</td>
<td>for create-react-app v5, we use craco to update the webpack config to polyfill some node modules</td>
</tr>
<tr>
<td>vanillademo js</td>
<td>source code (download)</td>
<td>uses a script tag to include a UMD bundle, and doesn’t require any transpilation or bundling</td>
</tr>
</tbody>
</table>

**JBrowse Jupyter**

Comprehensive documentation for JBrowse Jupyter is found off-site, here.
Also check out the JBrowse Jupyter GitHub repo.

**JBrowseR**

Comprehensive documentation for JBrowseR is found off-site, here.
Also check out the JBrowse Jupyter GitHub repo.
Command line tools

This document covers the CLI tools. Note: for @jbrowse/img static export tool, see https://www.npmjs.com/package/@jbrowse/img

Note: the @jbrowse/cli may not do all types of operations, some use cases may best be handled by creating your own tools to manipulate a config.json by hand or by using a script file.

A simple script that does not use @jbrowse/cli at all may just look like this

```javascript
const config = JSON.parse(fs.readFileSync('config.json', 'utf8'))
// do something with config.tracks, config.assemblies, etc.
fs.writeFileSync('config.json', JSON.stringify(config, null, 2))
```

Installation

The command line tools can be installed globally using npm as follows

```bash
$ npm install -g @jbrowse/cli
```

A CLI tool called jbrowse should then be available in the path. You can test your installation with

```bash
$ jbrowse --version
```

It is also possible to do one-off executions using npx, e.g.

```bash
npx @jbrowse/cli create myfolder
```

It is likely preferable in most cases to install the tools globally with npm install @jbrowse/cli -g however

Commands

- jbrowse add-assembly SEQUENCE
- jbrowse add-connection CONNECTIONURLORPATH
- jbrowse add-track TRACK
- jbrowse add-track-json TRACK
- jbrowse admin-server
- jbrowse create LOCALPATH
- jbrowse help [COMMAND]
- jbrowse remove-track TRACK
- jbrowse set-default-session
- jbrowse text-index
- jbrowse upgrade [LOCALPATH]

**jbrowse add-assembly SEQUENCE**

Add an assembly to a JBrowse 2 configuration

**USAGE**

```bash
$ jbrowse add-assembly SEQUENCE
```

**ARGUMENTS**

- **SEQUENCE**
  sequence file or URL

  If TYPE is indexedFasta or bgzipFasta, the index file defaults to <location>.fai
  and can be optionally specified with --faiLocation
  If TYPE is bgzipFasta, the gzip index file defaults to <location>.gzi
  and can be
optionally specified with --gziLocation

OPTIONS
-a, --alias=alias
   An alias for the assembly name (e.g. "hg38" if the name of the assembly is "GRCh38");
   can be specified multiple times

-f, --force
   Equivalent to `--skipCheck --overwrite`

-h, --help
   show CLI help

-l, --load=copy|symlink|move|inPlace
   Required flag when using a local file. Choose how to manage the data
directory. Copy, symlink, or move the data
directory to the JBrowse directory. Or use inPlace to modify the config
without doing any file operations

-n, --name=name
   Name of the assembly; if not specified, will be guessed using the
   sequence file name

-t, --type=indexedFasta|bgzipFasta|twoBit|chromSizes|custom
   type of sequence, by default inferred from sequence file

   indexedFasta  An index FASTA (e.g. .fa or .fasta) file;
   can optionally specify --faiLocation

   bgzipFasta    A block-gzipped and indexed FASTA (e.g. .fa.gz or
   .fasta.gz) file;
   can optionally specify --faiLocation and/or --gziLocation

   twoBit       A twoBit (e.g. .2bit) file

   chromSizes   A chromosome sizes (e.g. .chrom.sizes) file

   custom Either a JSON file location or inline JSON that defines a
custom
           sequence adapter; must provide --name if using inline JSON

--displayName=displayName
   The display name to specify for the assembly, e.g. "Homo sapiens (hg38)"
   while the name can be a shorter identifier
   like "hg38"

--faiLocation=faiLocation
   [default: <fastaLocation>.fai] FASTA index file or URL

--gziLocation=gziLocation
   [default: <fastaLocation>.gzi] FASTA gzip index file or URL

--out=out
   synonym for target
--overwrite
   Overwrite existing assembly if one with the same name exists

--refNameAliases=refNameAliases
   Reference sequence name aliases file or URL; assumed to be a
tab-separated aliases
   file unless --refNameAliasesType is specified

--refNameAliasesType=aliases|custom
   Type of aliases defined by --refNameAliases; if "custom",
   --refNameAliases is either
   a JSON file location or inline JSON that defines a custom sequence
   adapter

--refNameColors=refNameColors
   A comma-separated list of color strings for the reference sequence
   names; will cycle
   through colors if there are fewer colors than sequences

--skipCheck
   Don't check whether or not the sequence file or URL exists or if you are
   in a JBrowse directory

--target=target
   path to config file in JB2 installation directory to write out to.
   Creates ./config.json if nonexistent

EXAMPLES
# add assembly to installation in current directory. assumes .fai file also
   exists, and copies GRCh38.fa and
   GRCh38.fa.fai to current directory
$ jbrowse add-assembly GRCh38.fa --load copy

# add assembly to a specific jb2 installation path using --out, and copies
   the .fa and .fa.fai file to /path/to/jb2
$ jbrowse add-assembly GRCh38.fa --out /path/to/jb2/ --load copy

# force indexedFasta for add-assembly without relying on file extension
$ jbrowse add-assembly GRCh38.xyz --type indexedFasta --load copy

# add displayName for an assembly
$ jbrowse add-assembly myFile.fa.gz --name hg38 --displayName "Homo sapiens
   (hg38)"

# use chrom.sizes file for assembly instead of a fasta file
$ jbrowse add-assembly GRCh38.chrom.sizes --load inPlace

# add assembly from preconfigured json file, expert option
$ jbrowse add-assembly GRCh38.config.json --load copy

# add assembly from a 2bit file, also note pointing direct to a URL so no
   --load flag needed
$ jbrowse add-assembly https://example.com/data/sample.2bit

# add a bgzip indexed fasta inferred by fa.gz extension. assumes .fa.gz.gzi
   and .fa.gz.fai files also exists
$ jbrowse add-assembly myfile.fa.gz --load copy

See code: src/commands/add-assembly.ts

**jbrowse add-connection CONNECTIONURLORPATH**

Add a connection to a JBrowse 2 configuration

<table>
<thead>
<tr>
<th>USAGE</th>
<th>$ jbrowse add-connection CONNECTIONURLORPATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARGUMENTS</td>
<td>URL of data directory</td>
</tr>
<tr>
<td>CONNECTIONURLORPATH</td>
<td>For hub file, usually called hub.txt</td>
</tr>
<tr>
<td>For JBrowse 1, location of JB1 data directory similar to <a href="http://mysite.com/jbrowse/data/">http://mysite.com/jbrowse/data/</a></td>
<td></td>
</tr>
<tr>
<td>OPTIONS</td>
<td>Assembly name of the connection If none, will default to the assembly in your config file</td>
</tr>
<tr>
<td>-a, --assemblyName=assemblyName</td>
<td>Any extra config settings to add to connection in JSON object format, such as '({&quot;uri&quot;:&quot;url&quot;:&quot;<a href="https://sample.com%22%7D">https://sample.com&quot;}</a>, &quot;locationType&quot;: &quot;UriLocation&quot;)'</td>
</tr>
<tr>
<td>-c, --config=config</td>
<td>Equivalent to <code>--skipCheck --overwrite</code></td>
</tr>
<tr>
<td>-f, --force</td>
<td>show CLI help</td>
</tr>
<tr>
<td>-h, --help</td>
<td>Name of the connection. Defaults to connectionId if not provided</td>
</tr>
<tr>
<td>-n, --name=name</td>
<td>type of connection, ex. JBrowse1Connection, UCSCTrackHubConnection, custom</td>
</tr>
<tr>
<td>-t, --type=type</td>
<td>Id for the connection that must be unique to JBrowse. Defaults to ‘connectionType-assemblyName-currentTime’</td>
</tr>
<tr>
<td>--connectionId=connectionId</td>
<td>synonym for target</td>
</tr>
<tr>
<td>--out=out</td>
<td>Overwrites any existing connections if same connection id</td>
</tr>
<tr>
<td>--overwrite</td>
<td>Don’t check whether or not the data directory URL exists or if you are in a JBrowse directory</td>
</tr>
<tr>
<td>--skipCheck</td>
<td>path to config file in JB2 installation directory to write out to.</td>
</tr>
</tbody>
</table>

**EXAMPLES**

$ jbrowse add-connection http://mysite.com/jbrowse/data/

$ jbrowse add-connection http://mysite.com/jbrowse/custom_data_folder/ --type JBrowse1Connection
$ jbrowse add-connection http://mysite.com/path/to/hub.txt --assemblyName hg19
$ jbrowse add-connection http://mysite.com/path/to/custom_hub_name.txt --type UCSCTrackHubConnection --assemblyName hg19
$ jbrowse add-connection http://mysite.com/path/to/custom --type custom --config '{"uri":{"url":"https://mysite.com/path/to/custom"}, "locationType": "UriLocation"}' --assemblyName hg19
$ jbrowse add-connection https://mysite.com/path/to/hub.txt --connectionId newId --name newName --target /path/to/jb2/installation/config.json

See code: src/commands/add-connection.ts

**jbrowse add-track TRACK**

Add a track to a JBrowse 2 configuration

**USAGE**

```bash
$ jbrowse add-track TRACK
```

**ARGUMENTS**

- TRACK  Track file or URL

**OPTIONS**

- `-a`, `--assemblyNames=assemblyNames`  Assembly name or names for track as comma separated string. If none, will default to the assembly in your config file
- `-d`, `--description=description`  Optional description of the track
- `-f`, `--force`  Equivalent to `--skipCheck --overwrite`
- `-h`, `--help`  show CLI help
- `-l`, `--load=copy|symlink|move|inPlace`  Required flag when using a local file. Choose how to manage the track. Copy, symlink, or move the track to the JBrowse directory. Or inPlace to leave track alone
- `-n`, `--name=name`  Name of the track. Will be defaulted to the trackId if none specified
- `-t`, `--trackType=trackType`  Type of track, by default inferred from track file
- `--category=category`  Optional Comma separated string of categories to group tracks
- `--config=config`  Any extra config settings to add to a track. i.e. `{"defaultRendering": "density"}`
--indexFile [indexFile]
Optional index file for the track

--out [out]
synonym for target

--overwrite
Overwrites existing track if it shares
the same trackId

--protocol [protocol]
[default: uri] Force protocol to a
specific value

--skipCheck
Skip check for whether or not the file
or URL exists or if you are in a JBrowse
directory

--subDir [subDir]
when using --load a file, output to a
subdirectory of the target dir

--target [target]
path to config file in JB2
installation to write out to.

--trackId [trackId]
trackId for the track, by default
inferred from filename, must be unique
throughout config

EXAMPLES
# copy /path/to/my.bam and /path/to/my.bam.bai to current directory and adds
# track to config.json
$ jbrowse add-track /path/to/my.bam --load copy

# copy my.bam and my.bam.bai to /path/to jb2/bam and adds track entry to
# /path/to jb2/bam/config.json
$ jbrowse add-track my.bam --load copy --out /path/to jb2 --subDir bam

# same as above, but specify path to bai file. needed for if the bai file
does not have the extension .bam.bai
$ jbrowse add-track my.bam --indexFile my.bam --load copy

# creates symlink for /path/to/my.bam and adds track to config.json
$ jbrowse add-track /path/to/my.bam --load symlink

# add track from URL to config.json, no --load flag needed
$ jbrowse add-track https://mywebsite.com/my.bam

# --load inPlace adds a track without doing file operations
$ jbrowse add-track /url/relative/path.bam --load inPlace

See code: src/commands/add-track.ts

**jbrowse add-track-json TRACK**
Add a track configuration directly from a JSON hunk to the JBrowse 2 configuration

**USAGE**

```
$ jbrowse add-track-json TRACK
```

**ARGUMENTS**

```
TRACK  track JSON file or command line arg blob
```
OPTIONS
- update update the contents of an existing track, matched based on trackId
--out=out synonym for target
--target=target path to config file in JB2 installation directory to write out to.
Creates ./config.json if nonexistent

EXAMPLES
$ jbrowse add-track-json track.json
$ jbrowse add-track-json track.json --update

See code: src/commands/add-track-json.ts

jbrowse admin-server
Start up a small admin server for JBrowse configuration

USAGE
$ jbrowse admin-server

OPTIONS
-h, --help show CLI help
-p, --port=port Specified port to start the server on; Default is 9090.
--bodySizeLimit=bodySizeLimit [default: 25mb] Size limit of the update message; may need to increase if config is large.
Argument is passed to bytes library for parsing: https://www.npmjs.com/package/bytes.
--root=root path to the root of the JB2 installation. Creates ./config.json if nonexistent. note that you can navigate to ?config=path/to/subconfig.json in the web browser and it will write to rootDir/path/to/subconfig.json

EXAMPLES
$ jbrowse admin-server
$ jbrowse admin-server -p 8888

See code: src/commands/admin-server.ts

jbrowse create LOCALPATH
Downloads and installs the latest JBrowse 2 release

USAGE
$ jbrowse create LOCALPATH

ARGUMENTS
LOCALPATH  Location where JBrowse 2 will be installed

OPTIONS
- \( f, \) \( --force \)  
  Overwrites existing JBrowse 2 installation if present in path
- \( h, \) \( --help \)  
  show CLI help
- \( l, \) \( --listVersions \)  
  Lists out all versions of JBrowse 2
- \( t, \) \( --tag=tag \)  
  Version of JBrowse 2 to install. Format is v1.0.0. Defaults to latest
- \( u, --url=url \)  
  A direct URL to a JBrowse 2 release
- \( --branch=branch \)  
  Download a development build from a named git branch
- \( --nightly \)  
  Download the latest development build from the main branch

EXAMPLES
# Download latest release from github, and put in specific path
$ jbrowse create /path/to/new/installation

# Download latest release from github and force overwrite existing contents at path
$ jbrowse create /path/to/new/installation --force

# Download latest release from a specific URL
$ jbrowse create /path/to/new/installation --url url.com/directjbrowselink.zip

# Download a specific tag from github
$ jbrowse create /path/to/new/installation --tag v1.0.0

# List available versions
$ jbrowse create --listVersions

See code: src/commands/create.ts

jbrowse help [COMMAND]
display help for jbrowse

USAGE
$ jbrowse help [COMMAND]

ARGUMENTS
COMMAND command to show help for

OPTIONS
--all see all commands in CLI


jbrowse remove-track TRACK

Remove a track configuration from a JBrowse 2 configuration. Be aware that this can cause crashes in saved sessions that refer to this track!
USAGE
$ jbrowse remove-track TRACK

ARGUMENTS
TRACK track JSON file or command line arg blob

OPTIONS
--out=out synonym for target
--target=target path to config file in JB2 installation directory to write
   out to. Creates ./config.json if nonexistent

EXAMPLE
$ jbrowse remove-track-json trackId

See code: src/commands/remove-track.ts

jbrowse set-default-session
Set a default session with views and tracks

USAGE
$ jbrowse set-default-session

OPTIONS
-c, --currentSession List out the current default session
-h, --help show CLI help
-n, --name=name [default: New Default Session] Give a name for the
default session
-s, --session=session set path to a file containing session in json format
-t, --tracks=tracks Track id or track ids as comma separated string to
   put into default session
-v, --view=view View type in config to be added as default session,
   i.e LinearGenomeView, CircularView,
   DotplotView. Must be provided if no default session file provided
--delete Delete any existing default session.
--out=out synonym for target
--target=target path to config file in JB2 installation directory to
   write out to
--viewId=viewId Identifier for the view. Will be generated on default

EXAMPLES
$ jbrowse set-default-session --session /path/to/default/session.json
$ jbrowse set-default-session --target /path/to/jb2/installation/config.json
   --view LinearGenomeView --tracks track1,
   track2, track3
$ jbrowse set-default-session --view LinearGenomeView, --name newName
   --viewId view-no-tracks
$ jbrowse set-default-session --currentSession # Prints out current default session

See code: src/commands/set-default-session.ts

**jbrowse text-index**

Make a text-indexing file for any given track(s).

<table>
<thead>
<tr>
<th>USAGE</th>
<th>$ jbrowse text-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIONS</td>
<td></td>
</tr>
<tr>
<td>-a, --assemblies=assemblies</td>
<td>Specify the assembly(ies) to create an index for. If unspecified, creates an index for each assembly in the config</td>
</tr>
<tr>
<td>-h, --help</td>
<td>show CLI help</td>
</tr>
<tr>
<td>-q, --quiet</td>
<td>Hide the progress bars</td>
</tr>
<tr>
<td>--attributes=attributes</td>
<td>[default: Name, ID] Comma separated list of attributes to index</td>
</tr>
<tr>
<td>--dryrun</td>
<td>Just print out tracks that will be indexed by the process, without doing any indexing</td>
</tr>
<tr>
<td>--exclude=exclude</td>
<td>[default: CDS, exon] Adds gene type to list of excluded types</td>
</tr>
<tr>
<td>--file=fil</td>
<td>File or files to index (can be used to create trix indexes for embedded component use cases not using a config.json for example)</td>
</tr>
<tr>
<td>--fileId=fileId</td>
<td>Set the trackId used for the indexes generated with the --file argument</td>
</tr>
<tr>
<td>--force</td>
<td>Overwrite previously existing indexes</td>
</tr>
<tr>
<td>--out=out</td>
<td>Synonym for target</td>
</tr>
<tr>
<td>--perTrack</td>
<td>If set, creates an index per track</td>
</tr>
<tr>
<td>--prefixSize=prefixSize</td>
<td>[default: 6] Specify the prefix size for the ixx index, increase size if many of your gene IDs have same prefix e.g. Z0000000001, Z0000000002</td>
</tr>
<tr>
<td>--target=target</td>
<td>Path to config file in JB2 installation directory to read from.</td>
</tr>
<tr>
<td>--tracks=tracks</td>
<td>Specific tracks to index, formatted as comma separated trackIds. If unspecified, indexes all available tracks</td>
</tr>
</tbody>
</table>

**EXAMPLES**

# indexes all tracks that it can find in the current directory's config.json
$ jbrowse text-index

# indexes specific trackIds that it can find in the current directory's config.json
$ jbrowse text-index --tracks=track1,track2,track3

# indexes all tracks in a directory's config.json or in a specific config file
$ jbrowse text-index --out /path/to/jb2/

# indexes only a specific assembly, and overwrite what was previously there using force (which is needed if a previous index already existed)
$ jbrowse text-index -a hg19 --force

# create index for some files for use in @jbrowse/react-linear-genome-view or similar
$ jbrowse text-index --file myfile.gff3.gz --file myfile.vcfgz --out indexes

See code: src/commands/text-index.ts

**jbrowse upgrade [LOCALPATH]**

Upgrades JBrowse 2 to latest version

**USAGE**

$ jbrowse upgrade [LOCALPATH]

**ARGUMENTS**

LOCALPATH [default: .] Location where JBrowse 2 is installed

**OPTIONS**

- **-h, --help** show CLI help
- **-l, --listVersions** Lists out all versions of JBrowse 2
- **-t, --tag=tag** Version of JBrowse 2 to install. Format is v1.0.0.
  Defaults to latest
- **-u, --url=url** A direct URL to a JBrowse 2 release
- **--branch=branch** Download a development build from a named git branch
- **--clean** Removes old js, map, and LICENSE files in the installation
- **--nightly** branch Download the latest development build from the main branch

**EXAMPLES**

# Upgrades current directory to latest jbrowse release
$ jbrowse upgrade

# Upgrade jbrowse instance at a specific filesystem path
$ jbrowse upgrade /path/to/jbrowse2/installation

# Upgrade to a specific tag
$ jbrowse upgrade /path/to/jbrowse2/installation --tag v1.0.0
# List versions available on github
$ jbrowse upgrade --listVersions

# Upgrade from a specific URL
$ jbrowse upgrade --url https://sample.com/jbrowse2.zip

# Get nightly release from main branch
$ jbrowse upgrade --nightly

See code: src/commands/upgrade.ts

**Debugging**

Debug logs (provided by debug) can be printed by setting the `DEBUG` environment variable. Setting `DEBUG=*` will print all debug logs. Setting `DEBUG=jbrowse*` will print only logs from this tool, and setting e.g. `DEBUG=jbrowse:add-assembly` will print only logs from the `add-assembly` command.
FAQ

Developers

How can I start the JBrowse 2 app as a developer

We recommend that you have the following:

- A stable and recent version of node
- Git
- Yarn

Then you can follow steps from our README.

It basically boils down to:

```
git clone https://github.com/GMOD/jbrowse-components
cd jbrowse-components
yarn
cd products/jbrowse-web
yarn start
```

This will boot up a development instance of jbrowse-web on port 3000.

You can use `PORT=8080 yarn start` to manually specify a different port.

You can also instead go to the `products/jbrowse-desktop` directory to do this on desktop.

For the embedded components e.g. `products/jbrowse-react-linear-genome-view`, use `yarn storybook` instead of `yarn start`.

For a more extensive tutorial, see Developing with JBrowse web and desktop.

General

What is special about JBrowse 2

One thing that makes JBrowse 2 special is that we can create new view types via our plugin system, e.g. circular, dotplot, etc.. Anything you want can be added as a view, and can be shown alongside our other views.

This makes JBrowse 2 more than just a genome browser: it is really a platform that can be built upon.

What are new features in JBrowse 2

See the https://jbrowse.org/jb2/features page for an overview of features

Setup

What web server do I need to run JBrowse 2

JBrowse 2 by itself is just a set of JS, CSS, and HTML files that can be statically hosted on a webserver without any backend services running.

Therefore, running JBrowse 2 generally involves just copying the JBrowse 2 folder to your web server html folder e.g. copy `/var/www/html/` to Amazon S3.

If you use a different platform such as Django, you may want to put it in the static resources folder.

Note that the server that you use should support byte-range requests (e.g. the Range HTTP header so that JBrowse can get small slices of large binary data files.
BAM files do not work on my server

If you are using Apache then you will probably want to disable mime_magic. If mime_magic is enabled, you may see that your server responds with the HTTP header Content-Encoding: gzip which JBrowse does NOT want, because this instructs the browser to unzip the data but JBrowse should be in charge of this.

How can I setup JBrowse 2 on my web server

We recommend following the steps in the quickstart web via CLI guide.

The general procedure is using the `jbrowse create /var/www/html/jb2` and this will download the latest version of jbrowse to your web folder e.g. in `/var/www/html`.

You can also use `jbrowse upgrade /var/www/html/jb2` to get the latest version.

How do I install or update the @jbrowse/cli tool

To install the @jbrowse/cli tool, you can use `npm install -g @jbrowse/cli`.

You can use this same command to upgrade the tool too.

This command will give you a command named `jbrowse` which should automatically be in your path if you have a standard installation of nodejs. We recommend using nodesource or nvm to get your nodejs for this.

Also note that the @jbrowse/cli tool is just made for preparing your config.json, it is **not used to run any server-side code**.

How can I make a header on a jbrowse-web instance

You can edit the index.html that comes with jbrowse-web to have custom contents. The jbrowse-web app just looks at the div that it renders into, but any contents outside of that you can edit for custom purposes. If you need more advanced embedding, you can consider @jbrowse/react-linear-genome-view or similar, but the jbrowse-web app is not available as an npm installable package yet.

How do I update my instance of jbrowse-web

You can use the command, after installing:

```
jbrowse upgrade /path/to/your/jbrowse2
```

This will download the latest release from github and overwrite it onto your jbrowse-web instance.

If you’ve manually downloaded jbrowse-web, the newest releases can be found here.

How can I setup JBrowse 2 without the CLI tools

The jbrowse CLI tools are basically a convenience, and are not strictly required.

Simple tasks can be done without it.

For example, for jbrowse create, you can visit the releases page and download the latest jbrowse-web release tag, and unzip it into your web directory.

Checkout our quickstart web guide for a speedy start to using a manually downloaded JBrowse instance.

For other things, like add-assembly and add-track, you can manually edit the `config.json`; reviewing the config docs and sample configs will be valuable.

To configure JBrowse using the GUI, checkout our tutorial.

Understanding the config basics will come in handy also because you can manually edit in advanced configs after your tracks are loaded; however be careful: corrupt configs can produce hard to understand errors, because our config system is strongly typed.

Reach out to the team on gitter or in the discussions if you have any complex configuration issues.
How do I load a track into JBrowse 2

With the JBrowse CLI tools, you can easily add tracks with the add-track command, e.g.:

```
jbrowse add-track myfile.bw -a hg19
```

This will setup a bigwig track on the hg19 assembly in your config.json.

Make sure to run the command inside your current jbrowse2 folder e.g. /var/www/html/jbrowse2 or wherever you are currently setting up a config.json (you can have multiple configs).

Note that you can also use remote URLs

```
jbrowse add-track http://yourremote/myfile.bam
```

The add-track command will do as much as possible to infer from the file extension how to configure this track, and automatically infer the index to be myfile.bam.bai.

As mentioned above you can also manually edit your config file, or use the GUI.

How do I customize the color of the features displayed on my track

We use Jexl for defining configuration callbacks, including feature coloration.

An example of a Jexl configuration callback might look like this:

```
"color": "jexl: get(feature, 'strand') == -1? 'red': 'blue'"
```

See our configuration callbacks guide for more information.

My jexl is too complicated, how can I simplify it?

You can create a small plugin that adds a new function to the jexl language.

See here for an example of making a color callback.

Adding color callbacks in the GUI

In brief, to add a configuration callback to a track using the GUI, perform the following steps:

1. On the track you’re meaning to color, click on the three vertical dots ‘...’ on the right side of the track label
2. Click “Settings” (if this option is greyed out, copy the track with “Copy Track”, then open up the track under “Session Tracks” and repeat steps 1-2)
3. Scroll down to the “display 1 renderer” heading (this is typically the display you want to edit, if not scroll to display 2)
4. Click on the circle to the right of the color you’d like to change
5. In this text box, enter in the Jexl callback for the feature colouration, e.g. “get(feature,'strand') == -1 ? ‘red’ : ‘blue’ ”

Adding color callbacks via the command line

Adding color callbacks via the command line can be a little tricky because the coloration property exists within the renderer.

In brief, to add a configuration callback to a track using the CLI, your add-track is going to look something like this:

```
jbrowse add-track somevariants.vcf --load copy --config '{"displays": [{"displayId": "my_BasicDisplay", "type": "LinearBasicDisplay", "renderer": {"color1": "jexl: get(feature, '\"strand\"') == -1 ? '\"red\"' : '\"blue\"" }]}]}'
```
While adding the track to the `config.json`, you’re adding additional configurations using the `--config` option. This additional configuration is a “renderer” on the display that your track will be using. In this case, this `.vcf` will be using the `LinearBasicDisplay`.

**Curiosities**

**Why do all the tracks need an assembly specified**

We require that all tracks have a specific genome assembly specified in their config. This is because JBrowse 2 is a multi-genome-assembly browser (and can compare genomes given the data). This may be different to using, say, JBrowse 1 where it knows which genome assembly you are working with at any given time.

**How are the menus structured in the app**

In JBrowse 1, the app level menu operated on the single linear genome view, but with JBrowse 2, the top level menu only performs global operations and the linear genome view has its own hamburger menu. Note that each track also has its own track level menu.

**Why do some of my reads not display soft clipping**

Some reads, such as secondary reads, do not have a SEQ field on their records, so they will not display softclipping. The soft clipping indicators on these reads will appear black.

**Do you have any tips for learning React and mobx-state-tree**

Here is a short guide to React and mobx-state-tree that could help get you oriented:

https://gist.github.com/cmdcolin/94d1cb285e6319cc3af4b9a8556f03f

**What technologies does JBrowse 2 use**

We build on a lot of great open source technology, some main ones include:

- React
- mobx-state-tree
- web-workers
- Typescript
- Electron (for desktop specifically)

**Should I configure gzip on my web server**

Yes! JBrowse 2 may load ~5MB of JS resources (2.5MB for main thread bundle, 2.5MB for worker bundle). If you have gzip enabled, the amount of data the user has to download though is only 1.4MB. We have worked on making bundle size small with lazy loading and other methods but adding gzip will help your users.

It will depend on your particular server setup e.g. apache, nginx, cloudfront, etc. how this may be done, but it is recommended to look into this.

**How does JBrowse know when to display the “Zoom in to see more features” message**

The rules that JBrowse uses to determine when to display the “Zoom in to see more features” message are called stats estimation rules.

The general outline is:

- It doesn’t display a zoom in message if zoomed in closer than 20kb
- It performs byte size estimation for BAM and CRAM type files (you will see a byte size estimation displayed alongside the “Zoom in to see features” message
- Other data types that don’t use byte size estimation use feature density based calculation
- Hi-C, BigWig, and sequence adapters are hardcoded to return `{ featureDensity:0 }` to always render
If you need to customize your particular track, you can set config variables on the “display” section of your config:

- **maxFeatureScreenDensity**: number of features times bpPerPx
- **fetchSizeLimit**: this config variable exists on the adapters (can increase size limit)

Example config with a small feature screen density:

```json
{
    "type": "VariantTrack",
    "trackId": "variant_density",
    "name": "test variants (small featuredensity limit)",
    "assemblyNames": ["volvox"],
    "adapter": {
        "type": "VcfTabixAdapter",
        "vcfGzLocation": {
            "uri": "volvox.filtered.vcf.gz"
        },
        "index": {
            "location": {
                "uri": "volvox.filtered.vcf.gz.tbi"
            }
        }
    },
    "displays": [
        {
            "type": "LinearVariantDisplay",
            "maxFeatureScreenDensity": 0.0006,
            "displayId": "volvox_filtered_vcf_color-LinearVariantDisplay"
        }
    ]
}
```

Example config for a CRAM file with a small fetchSizeLimit configured:

```json
{
    "type": "AlignmentsTrack",
    "trackId": "volvox_cram",
    "name": "test track (small fetch size limit)",
    "assemblyNames": ["volvox"],
    "adapter": {
        "type": "CramAdapter",
        "cramLocation": {
            "uri": "volvox-sorted-altname.cram"
        },
        "craiLocation": {
            "uri": "volvox-sorted-altname.cram.crai"
        },
        "sequenceAdapter": {
            "type": "TwoBitAdapter",
            "twoBitLocation": {
                "uri": "volvox.2bit"
            }
        },
        "fetchSizeLimit": 1000
    }
}
```
Text searching

Why I am running out of disk space while trix is running

The jbrowse text-index program will output data to a TMP directory while indexing. If your filesystem has low diskspace for /tmp you can set an alternative temporary directory using the environment variable TMPDIR=-/alt_tmp_dir/ jbrowse text-index.

How does the jbrowse text-index trix format work

The jbrowse text-index command creates text searching indexes using trix. The trix indexes are based on the format described by UCSC here https://genome.ucsc.edu/goldenPath/help/trix.html, but we re-implemented the code the create these index formats in the JBrowse CLI so you do not have to install the UCSC tools.

The main idea is that you give trix:

```
GENEID001  Wnt  signalling
GENEID002  ey  Pax6
```

Then this will generate a new file, the .ix file, sorted in alphabetical order:

```
ey  GENE002
signalling  GENE001
Pax6  GENE002
Wnt  GENE001
```

Then a second file, the .ixx file, tells us at what byte offset certain lines in the file are e.g.:

```
signa000000435
```

Note that JBrowse creates a specialized trix index also. Instead of creating a ix file with just the gene names, it also provides their name and location in an encoded format.

URL params

Why can’t I copy and paste my URL bar to share it with another user

In JBrowse Web, the current session can become too long to store in the URL bar, so instead, we store it in localStorage and only keep the key to the localStorage entry in the URL var. This is because otherwise URLs can get prohibitively long, and break server side navigations, intermediate caches, etc. Therefore, we make “sharing a session” a manual step that generates a shortened URL by default.

Note 1: users of @jbrowse/react-linear-genome-view have to re-implement any URL query param logic themselves, as this component makes no attempt to access URL query params.

Note 2: You can copy and paste your URL bar and put it in another tab on your own computer, and JBrowse will restore the session using BroadcastChannel (supported on Firefox and Chrome).

How does the session sharing work with shortened URLs work in JBrowse Web

We have a central database hosted as a AWS dynamoDB that stores encrypted session snapshots that users create when they use the “Share” button. The “Share” button creates a random key on the client side (which becomes the &password= component of the share URL), encrypts the session client side, and sends the encrypted session without the key to the AWS dynamoDB.

This process, generates a URL with the format:

```
&session=share-<DYNAMODBID>&password=<DECODEKEY>
```

The DECODEKEY is never transmitted to the server, but you can copy and paste the share URL, the person you shared automatically downloads the DynamoDB entry, and decodes it with the DECODEKEY from the URL that you provide

With this system, the contents of the dynamoDB are safe and unable to be read, even by JBrowse administrators.
Troubleshooting

Doing things like:

- Changing trackIds
- Deleting tracks

Can make user’s saved sessions fail to load. If part of a session is inconsistent, currently, the entire session will fail to load. Therefore, make decisions to delete or change IDs carefully.

What should I do if the Share system isn’t working?

If for any reason the session sharing system isn’t working, e.g. you are behind a firewall or you are not able to connect to the central share server, you can click the “Gear” icon in the “Share” button pop-up, and it will give you the option to use “Long URL” instead of “Short URL” which lets you create share links without the central server.

Also, if you are implementing JBrowse Web on your own server and would like to create your own URL shortener, you can use the shareURL parameter in the config.json file to point at your own server instead of ours.