Need-To-Know for

pal Development

Stacey Beard

Git Version Control

Branching

- Opal development is organized in the following development stages:

Personal branch	Team branch	Staging	PreProd	Prod
• Used by one person for development.	• **Optional** • If working in a team, can be used to merge everyone's personal branches to test them together.	• Once a project is done, a pull request to staging is created. • After code review, the project is merged into staging. • This branch is built into the Opal Staging app (for developers).	• Once staging has been tested, it's merged into PreProd. • PreProd is built into the Opal-dev app, then tested on mobile devices. • Only developers have access to this app.	• Once PreProd has been tested, it's merged into Prod. • Prod is built into the Opal app, then tested one last time on mobile devices. • After testing, a new version of Opal is released to patients.

Branching

- We use the following branch names for each development stage (underscores should be used to divide words):

Personal branch	Team branch	Staging	PreProd	Prod
Your project name, followed by an underscore, followed by your first name, all in lowercase e.g.: **questionnaires_anna**	The project name, in lowercase e.g.: **questionnaires**	Branch name: **staging**	Branch names: **opal_pre_prod** (qplus) **PreProd** (opal-listener)	Branch name: **master**

Note: for non-projects (ex: small changes, bug fixes), a short descriptive name (e.g. update_bootstrap) or issue number (e.g. issue_180) can be used.

Your Personal Branch

- During your opal project, you will only be working in the **personal branch** and **team branch** stages (don't touch the opal_pre_prod or master branches). You won't need a team branch if you're the only one working on your project.
- You'll interact with the **staging** branch only via pull request.
- Create a new personal branch:
  - Branch off of staging (to start a new project) or an existing project branch (to continue an existing project)
  - Branch name: your project name, followed by an underscore, followed by your first name, all in lowercase
  - e.g.: questionnaires_anna

Bug Fixing

- If you encounter a bug *that's not specifically related to your project*, and want to fix it, it's good practice to fix it on a separate branch.
  - **Why?** This will ensure that the fix is available to everyone as quickly as possible.
- When fixing bugs, use a separate branch for each bug (branch off of staging).
- Exception:
  - You should fix multiple bugs on the same branch when it makes sense to do so (ex: they are related, or in the same area of code).
- When you're done, submit a pull request to staging.

Commits

- Make commits to your branch after each logical change.
- To keep your code safe, push your branch to the origin (using the same branch name) regularly.

Commit style:

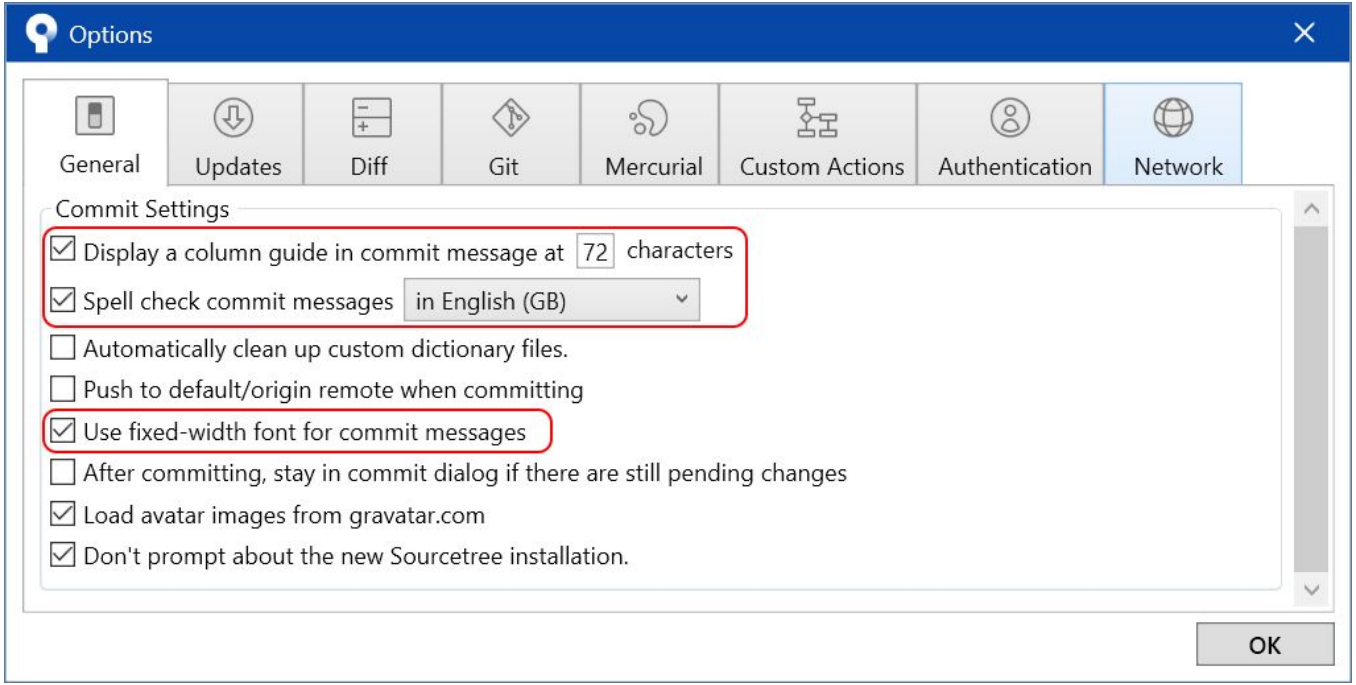
Make informative commits, using good commit style. A new student or developer who continues your work later should be able to understand what you've done.
Follow this commit guide: *How to Write a Git Commit Message*

Sourcetree Configurations

- Sourcetree can be configured to help you write better commit messages.

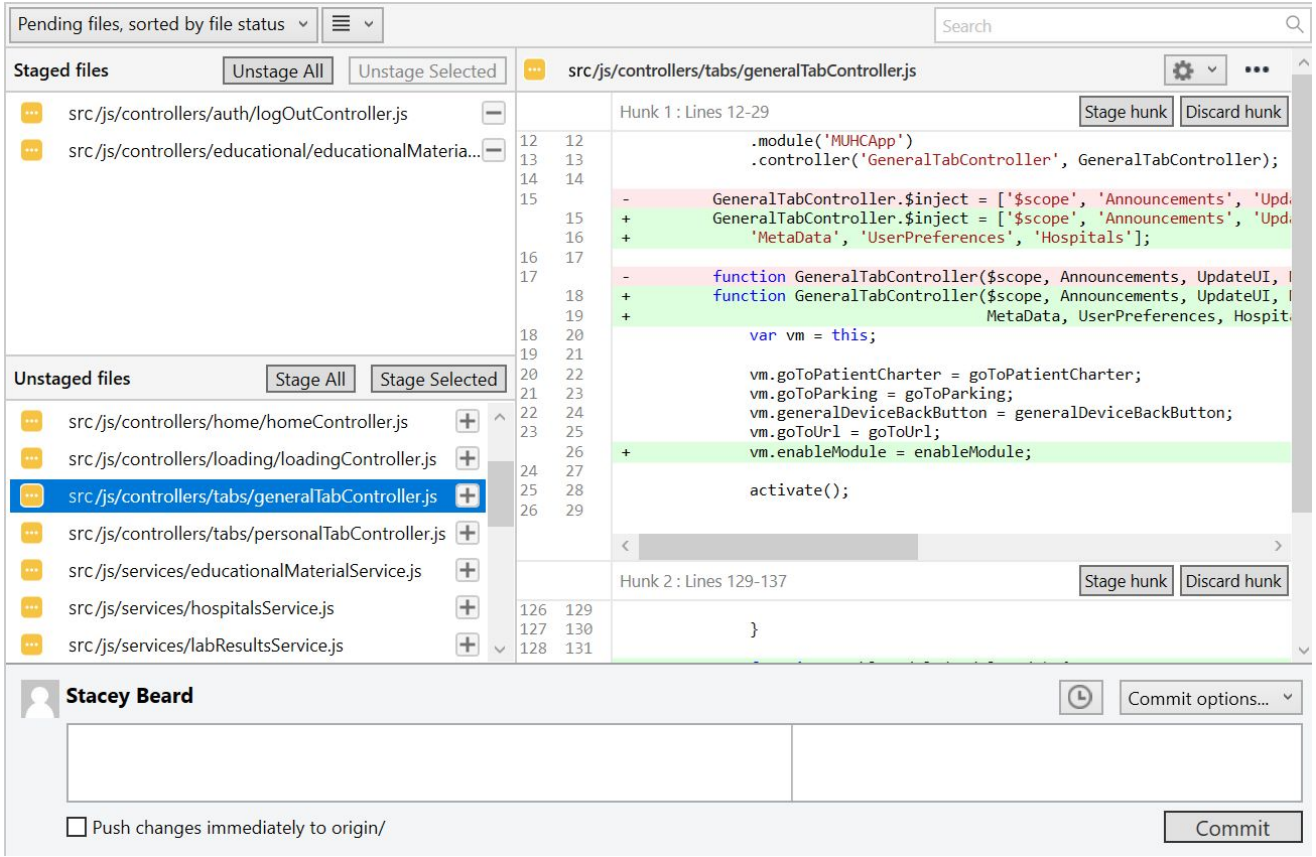
Go to **Tools > Options** and scroll down

Select these options →



Committing

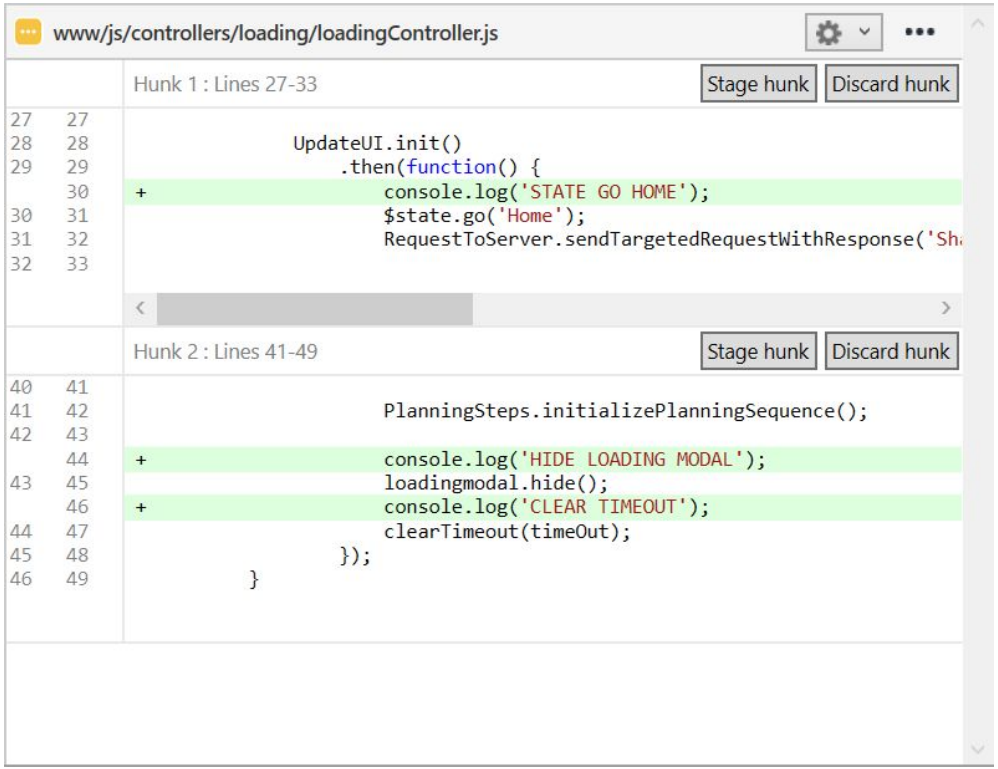
- **Always double-check your code changes before commiting.**
- A GUI like SourceTree can help you do this.



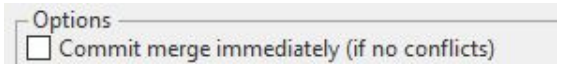
Committing

- Don't commit:
  - Personal configurations (except opal.config.js)
  - Shortcuts (like hardcoded email and password)
  - Console.logs used for debugging

Don't commit this! →



Keeping Your Branches Up to Date

- While working on your project, you should always **keep your branch up to date** with staging.
- Regularly merge the staging branch back into your project to get the latest changes.
- Don't commit these merges immediately:
  - 

  - ```
    git merge staging --no-commit
    ```
  - This will allow you to check for conflicts before completing the merge. This is easiest to do in SourceTree. Commit when you're done checking all the new changes.
- Always do this before creating a pull request (including for bug fixes).
    ```

Stashing

- Stashing can be a great way to save code that you want to put aside for later without having to commit it.
- Read more here: Git Stash
- Sourcetree provides a nice interface for managing your stashes (they're all listed in the left menu panel).
- Always stash leftover changes before switching branches to make sure you don't lose them.

WebStorm (or equivalent IDE)

Useful Features

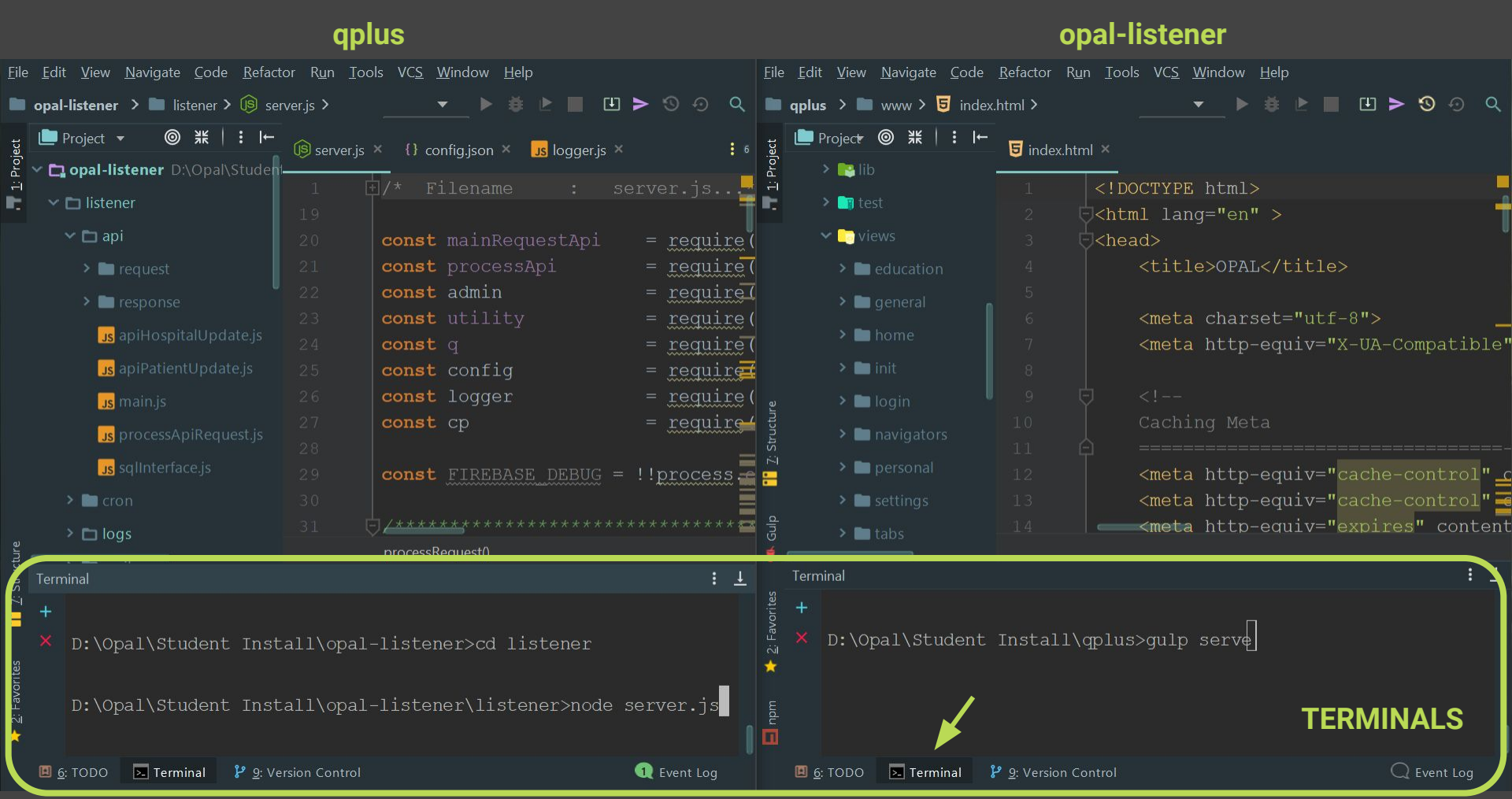
- Webstorm offers many useful features that can make development easier. Similar IDEs may have similar features, using different key combinations.

Features

- Global search in a project. You can specify a custom scope or specific file types.
- CTRL+Click (⌘+Click) on a function call to open the function in the file where it's defined.
- CTRL+Shift+N (Shift+⌘+O) to search for a file by name.
- And many more.

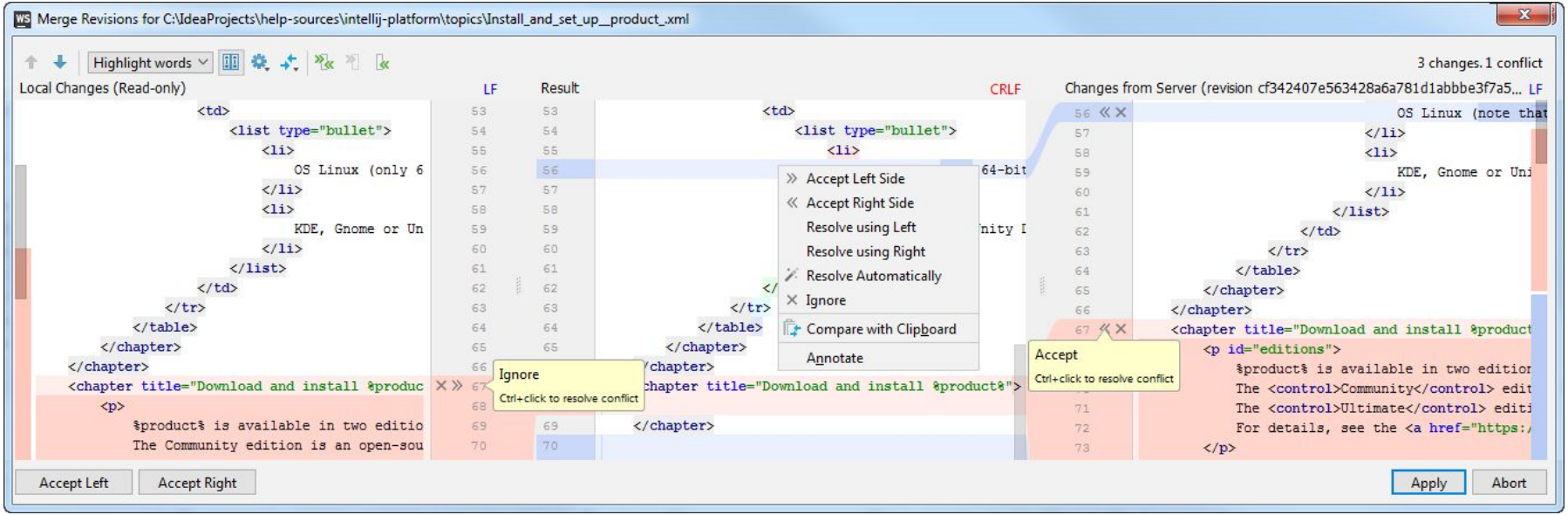
Terminal

- When running your local development environment, you can open your two projects (qplus and opal-listener) in two separate windows. Each will have a terminal where you can run the code.


qplus
opal-listener
File Edit View Navigate Code Refactor Run Tools VCS Window Help
opal-listener > listener > server.js
Project
opal-listener D:\Opal\Student
listener
api
request
response
apiHospitalUpdate.js
apiPatientUpdate.js
main.js
processApiRequest.js
sqlInterface.js
cron
logs
server.js
config.json
logger.js
/* Filename : server.js...
const mainRequestApi = require(
const processApi = require(
const admin = require(
const utility = require(
const q = require(
const config = require(
const logger = require(
const cp = require(
const FIREBASE_DEBUG = !!process
processRequest()
Terminal
D:\Opal\Student Install\opal-listener>cd listener
D:\Opal\Student Install\opal-listener\listener>node server.js
6: TODO
Terminal
9: Version Control
Event Log
qplus > www > index.html
Project
lib
test
views
education
general
home
init
login
navigators
personal
settings
tabs
index.html
<!DOCTYPE html>
<html lang="en" >
<head>
<title>OPAL</title>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible"
<!--
Caching Meta
<meta http-equiv="cache-control"
<meta http-equiv="cache-control"
<meta http-equiv="expires" content
Terminal
D:\Opal\Student Install\qplus>gulp serve
TERMINALS
6: TODO
Terminal
9: Version Control
Event Log

Merge Tool

- WebStorm has a great tool to help you fix merge conflicts.
- Read this to learn how to use it: [WebStorm - Resolving Conflicts]

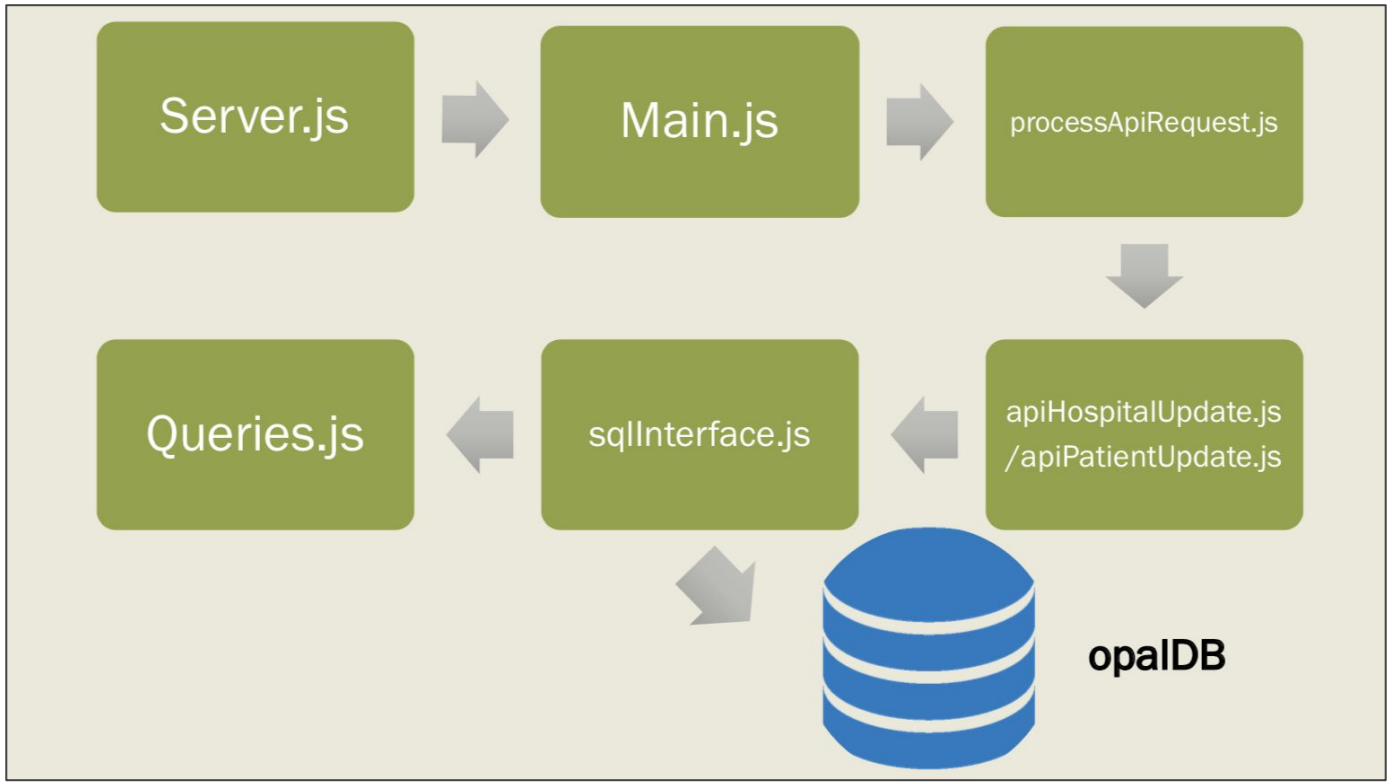


Opal Listener

Information Flow

- To understand how the listener works, start at `server.js` and read through the code.
- `server.js` is the file that runs when starting the listener with the command `node server` (the `.js` is implied).

Information Flow



API

- All requests are listed in `processApiRequest.js`
  - `API` contains all authenticated requests (made after the patient has logged in).
  - `securityAPI` contains all unauthenticated requests (made when the patient is not logged in).

Authenticated

```
const API = {
    'DeviceIdentifier': apiHospitalUpdate.updateDeviceIdentifier,
    'Log': apiPatientUpdate.logActivity,
    'LogPatientAction': apiPatientUpdate.logPatientAction,
    'Login': apiPatientUpdate.login,
    'Logout': apiHospitalUpdate.logout,
    'Resume': apiPatientUpdate.resume,
    'Refresh': apiPatientUpdate.refresh,
    'AccountChange': apiHospitalUpdate.accountChange,
    'CheckCheckin': apiPatientUpdate.checkCheckin,
    'Checkin': apiHospitalUpdate.checkIn,
    'CheckinUpdate': apiPatientUpdate.checkinUpdate,
    'DocumentContent': apiPatientUpdate.getDocumentsContent,
    'Feedback': apiHospitalUpdate.inputFeedback,
    'LabResults': apiPatientUpdate.getLabResults,
    'MapLocation': apiPatientUpdate.getMapLocation,
    'Message': apiHospitalUpdate.sendMessage,
    'NotificationsAll': apiHospitalUpdate.getAllNotifications,
    'NotificationsNew': apiHospitalUpdate.getNewNotifications,
    'EducationalPackageContents': apiPatientUpdate.getPackageContents,
    'Questionnaires': apiPatientUpdate.getQuestionnaires,
    'QuestionnaireRating': apiHospitalUpdate.inputEducationalMaterialRating,
    'QuestionnaireAnswers': apiHospitalUpdate.inputQuestionnaireAnswers,
    'Read': apiHospitalUpdate.updateReadStatus,
    'PFPMembers': apiPatientUpdate.getPatientsForPatientsMembers,
};
```

Unauthenticated

```
exports.securityAPI = {
    'PasswordReset': security.resetPasswordRequest,
    'SecurityQuestion': security.securityQuestion,
    'SetNewPassword': security.resetPasswordRequest,
    'VerifyAnswer': security.resetPasswordRequest
};
```

Adding a New Request

- When working on an Opal project, you will likely need to add new requests to the listener.
- To add a new request:
  - Give it an appropriate name, and add it to the API in `processApiRequest.js` (most of the time, you won't need to use the security API).
  - Create a function in `apiHospitalUpdate.js` or `apiPatientUpdate.js` for your request.
  - Create one or many functions in `sqlInterface.js` for your request. This is where the bulk of the request work gets done.
  - Add any DB queries to `queries.js`.

Adding a New Request

- After these steps, link all the functions together by using the same data flow as the existing Opal requests
  - `processApiRequest.js` calls `apiHospitalUpdate.js` or `apiPatientUpdate.js`, which calls `sqlInterface.js`, which uses the function `runSqlQuery` to run queries from `queries.js`
- When creating new queries, use question marks as placeholders for variables. **Never** assemble queries using string concatenation! This makes them vulnerable to SQL injection attacks.
- The easiest way to do this is to imitate what is being done in other Opal requests.

Do's and Don'ts of New Requests

Do

- Add new requests when needed.
- Edit new requests that were created as part of the project you're working on (by previous contributors).
- Use parameters to make the most of a single request.
- Split the requests you want to create into logical, single-purpose requests.

Don't

- Edit existing requests (talk to an Opal team member first if you think this is necessary).
- Delete existing requests.
- Make many separate requests that do similar things (use parameters instead).
- Make one large request that does many different things.

New Listener Structure

- A new structure for adding requests to the listener was introduced by David in April 2020.
- Read more about this structure here: https://github.com/Sable/opal-listener/pull/7
- This new structure is a bit more complicated to understand, but is cleaner and better.
- If possible, try to learn this new structure and use it, especially if your project involves creating a brand new module for the app.

Debugging in the Listener

- Debugging in the listener can be challenging.
  - The console outputs hundreds and hundreds of long lines.
  - Sometimes, what you want to look at gets cut off the top of the terminal because the output is too long.
- Solution: use the listener logs to send all output to a file.

Logger.js file →

logger.js

```
const levels = {
    error: 0,
    warn: 1,
    info: 2,
    verbose: 3,
    debug: 4,
};

winston.configure({
    transports: [
        // Output all debug events and higher to console
        new (winston.transports.Console)({
            level: 'debug',
            json: false,
            timestamp: true
        }),
        // Store all ---debug--- events and higher to the opal.log file
        new (winston.transports.File)({
            filename: './logs/opal-info.log',
            level: 'debug',
            json: true,
            timestamp: true
        })
    ]
});
```

Each logger message is similar to a console output, with the addition of a level that specifies its level of importance (error is the most important; debug is the least)

This section sends logs to an output file. Change the level to debug to send all logs (of level debug and higher, which is all of them) to a file

Important: don't commit this change!

Debugging in the Listener

- After saving and restarting the listener, all logger messages will get sent to the file opal-info.log, in the listener's logs folder.
- CAUTION: This file will become huge very fast (hundreds of megabytes; tens of thousands of lines).
  - Windows Notepad and WebStorm can't handle big text files with long lines. I recommend using Notepad++ or a similar advanced text editor to read the log instead.
  - It's safe to delete this file whenever it gets too big (this also makes it easier to find what you're looking for); the file will simply get re-created automatically.

Debugging in the Listener

- To debug the listener:
  - Stop the listener.
  - Make the change to the logger.js file.
  - Delete the existing opal-info.log file.
  - Restart the listener.
  - Use the Opal frontend to repeat the steps that created the issue.
  - Stop the listener (to avoid new logs being added while you're searching in the file).
  - Open opal-info.log using an advanced text editor like Notepad++. Search through the log for errors. You may want to toggle between word-wrap and no word-wrap to find what you're looking for.

Debugging in the Listener

- If you can't find the problem, you can add logger outputs of your own to print out variables. Make sure to use JSON.stringify(obj) when printing objects.

Example of logging (the first argument is the logging level, the second argument is the string to log):

```
logger.log('debug', 'results: ' + JSON.stringify(results));
```

It's good to add logger output to your code, just make sure to pick the appropriate logging level.

Opal Frontend

Requests

- Requests are sent to the listener via the service "RequestToServer".
- To send a request, use the RequestToServer API:

Examples:

```
RequestToServer.sendRequest('Read',{
    'Id':serNum,
    'Field':'Announcements'
});

RequestToServer.sendRequestWithResponse('LabResults')
    .then(function (response) {
```

Webpack

- Webpack is a module bundler used in qplus to build and serve the app.
- It does many useful things:
  - Module importing
  - Bundling
  - Building
  - DevServer (with auto-reload)
  - Minification
  - Environment management
  - Integration with npm dependency management
  - etc.
- To read more about webpack's use in qplus, refer to: https://github.com/Sable/qplus/pull/722

Npm Dependencies

- Npm dependencies (specified in package.json) provide access to third-party packages.
- Do not update npm dependencies unless it's necessary for your project. Many new versions of dependencies introduce breaking changes that will cause your Opal installation to stop working.
- If you need to install a new dependency for your project:
  - Always use the **--save** or **--save-dev** flag to record it in package.json.
  - Commit package.json **and** package-lock.json after installing the new dependency.
  - Document the new dependency in as many places as possible (especially in your pull request)! This will tell others that they need to run npm install before running your code.

Running Two or More Instances of qplus at Once

- It's sometimes useful for development to have two instances of the app running at the same time.
- For example, you can be logged in as a patient in one instance and as a caregiver in the other, to instantly see the results of sharing data.
- You can run two or more instances of qplus by using different ports (in two or more terminals):

```
npm run start
npm run start -- --port=9002
```

The first instance will use port 9001 (by default, from webpack.config.js).
The second instance will bypass the default to use port 9002.

JavaScript

Missing Attributes

- Always check that the attributes you're referencing exist.

Example:

```
if (user.firstName === "Jane")
```

Bad! This will crash if the user has no firstName attribute

Missing Attributes

- Always check that the attributes you're referencing exist.

Example:

```
if (user.hasOwnProperty('firstName')
    && user.firstName === "Jane")
```

Good! The second check won't be evaluated if the first one evaluates to false (if `'firstName'` is missing).

Undefined Attributes

- There are several ways of checking if attributes are undefined

Check for falsiness:

```
if (user.hasOwnProperty('firstName')
     && !user.firstName) // Empty first name
```

Caution! This will work for strings or objects, but the number 0 will evaluate to false, which may not be the behavior you want.

Undefined Attributes

- There are several ways of checking if attributes are undefined

Check typeof:

```
if (user.hasOwnProperty('firstName')
     && typeof user.firstName === 'undefined') …
```

Good!

Other methods: Checking for undefined

Documentation

Opal Documentation

- Document everything you're doing during your project in a Google document. This can include:
  - Description or introduction to your project.
  - Choices that you made while designing your project, and their justifications.
  - Evaluations of pros and cons.
  - GUI mockups of views.
  - Implementation details (what you did, and how).
  - List of known bugs / things that are incomplete in your project.
  - Need-to-Know for future development.
  - Suggestions of future directions.

Opal Documentation

Detailed documentation is the best way to guarantee that your project will live on after you.

"What do I include?" It's simple: include everything that you would want to know if you were a new person picking up your project where you left off.

End of Need-To-Know for Opal Development